UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería en Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

Desarrollo de un sistema de monitorización para SDNs (Software Defined Networks)

María Teresa Pegado Boureghida
Tutor: Francisco Javier Ramos de Santiago
Ponente: Jorge López de Vergara Méndez

Julio 2015

Desarrollo de un sistema de monitorización para SDNs (Software Defined Networks)

AUTOR: María Teresa Pegado Boureghida
TUTOR: Francisco Javier Ramos de Santiago

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2015

Resumen

Las redes actuales están constituidas por un conjunto heterogéneo de tecnologías y protocolos que dificulta en gran medida el despliegue, mantenimiento y monitorización de las mismas. Adicionalmente, nuevos servicios como la virtualización de infraestructuras en entornos cloud (Infrastructure as a Service o IaaS) o la virtualización de servicios de red (Network Function Virtualization) están creciendo masivamente complicando aún más las tareas de mantenimiento y despliegue de las redes de comunicaciones.

En los últimos años ha surgido el concepto SDN (Software Defined Network) que representa un nuevo paradigma dentro de las redes de comunicaciones. SDN se basa en la separación entre los planos de control y datos en los equipos de red haciendo que la gestión, evolución y funcionamiento de los mismos se simplifique añadiendo aproximaciones software centralizadas.

Aprovechando está simplificación que proporcionan las SDN, en este trabajo se describe el desarrollo de un sistema de monitorización para SDN, que permita aplicar diversas políticas a la red, mediante reglas instalables en los switches a través de un controlador; para la comunicación entre el controlador y los switches se empleará el protocolo OpenFlow.

Palabras clave

SDN, OpenFlow, monitorización, redes de comunicaciones.

Abstract

Current networks present a heterogeneous set of technologies and protocols that greatly hinder the deployment, maintenance and monitoring of the same. Moreover, new services as Cloud virtualization (Infrastructure as a Service or IaaS) or Network Function Virtualization (NFV) is growing massively further complicating maintenance and deployment of communication networks.

In the last years the concept of SDN (Software Defined Network) which represent a new paradigm in network communications has emerged. SDN is based in the separation between control and data planes in the network devices easing the management, evolution and performance thereof by adding centralized software approaches.

Taking advantage of this simplification that SDN provides ,in this work it is described the development of a Monitoring System for SDN that allows the application of different network policies, by means of installable rules on the switches through OpenFlow protocol.

Keywords

SDN, OpenFlow, Monitoring, communication networks.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Actualmente, las redes de ordenadores están evolucionando tanto en velocidad como en complejidad lo cual está motivando un cambio en las arquitecturas y paradigmas tradicionales. Muchas redes convencionales son jerárquicas, construidas con niveles de conmutadores Ethernet dispuestos en una estructura de árbol. Este diseño tenía sentido cuando la estructura cliente-servidor era dominante. A día de hoy, una arquitectura estática no se adapta bien a las necesidades dinámicas de computación y almacenamiento de las empresas, centros de datos y entornos académicos actuales.

En los centros de proceso de datos de las empresas, los patrones de tráfico han cambiado significativamente. A diferencia de las aplicaciones cliente-servidor donde se produce la mayor parte de la comunicación entre un cliente y un servidor, las aplicaciones de hoy en día acceden a diferentes bases de datos y servidores, creando una ráfaga de tráfico máquina a máquina antes mandar los datos al dispositivo final. Además con el desarrollo de tecnologías BigData, el paradigma de comunicación y computación ha cambiado hacia un modelo completamente distribuido que requiere un procesamiento paralelo masivo en miles de servidores conectados entre sí. Además, el aumento de los mega-conjuntos de datos causa una demanda constante de capacidad adicional en la red de los centros de datos.

Por otro lado, los usuarios emplean cada vez más dispositivos móviles personales, como teléfonos inteligentes, tabletas y portátiles para el acceso a la red corporativa. Este fenómeno se trata de una política empresarial denominada BYOD (Bring Your Own Device). Esto puede ser más cómodo para los empleados pero puede ser perjudicial para la empresa ya que puede dejar fisuras donde se puede filtrar la información o introducir aplicaciones maliciosas en la red. Gestionar todos estos accesos en las redes actuales proporcionando seguridad y rendimiento para la empresa es una tarea difícil que ha complicado en gran medida las labores de los gestores de red, como se explica en [1].

Además, existe una creciente tendencia para virtualizar infraestructuras en entornos Cloud (Infrastructure as a Service) [2]. Proporcionar este servicio, ya sea en una nube privada, mixta o pública, requiere una computación escalable y elástica, almacenamiento y recursos de red, idealmente con un punto de vista común y herramientas comunes.

Adicionalmente, debido a la gran complejidad de las redes actuales, la monitorización del tráfico se ha convertido en una tarea fundamental para asegurar la calidad y seguridad de los servicios y sistemas que hacen uso de las redes de comunicaciones. En este sentido, el desarrollo de herramientas de monitorización que sean flexibles, efectivas y que produzcan datos abiertos para los analistas y gestores de red es una tarea crucial para sacar el máximo de las redes actuales.

En definitiva, cumplir con los requisitos actuales del mercado es prácticamente imposible con la arquitectura de red tradicional. Las empresas están tratando de exprimir al máximo sus redes usando herramientas de gestión de alto nivel y procesos manuales. Las arquitecturas de red existentes no fueron diseñadas para cumplir con los requisitos de los

usuarios de hoy en día, las empresas, y los operadores; más bien los arquitectos de redes están condicionados por las limitaciones de las redes actuales.

Para enfrentarse a estos retos, en los últimos años se ha desarrollado la arquitectura de red SDN con el fin de facilitar la gestión y dar cabida a todas estas nuevas tendencias. La arquitectura SDN se basa en la separación de los planos de control y datos otorgando una visión global de la red y permitiendo un control centralizado de la misma.

1.1.1 Limitaciones de las redes actuales

Una de las principales limitaciones es la complejidad de las redes actuales. Los protocolos actuales de red proporcionan un mayor rendimiento y fiabilidad, conectividad más amplia, y seguridad más estricta. El problema es que cada protocolo da solución a un problema específico, sin el beneficio de ningún tipo de abstracción.

La naturaleza estática de las redes está en marcado contraste con la naturaleza dinámica del entorno de los centros de procesos de datos, donde la virtualización altera la localización física supuesta de los hosts y provoca un incremento del número de hosts que requieren conectividad de red. Debido a su naturaleza estática y poco reconfigurable, la red no puede adaptarse dinámicamente a las demandas cambiantes del tráfico, aplicaciones y usuarios. La complejidad y la naturaleza dinámica de la red, hace que la configuración y gestión de la misma sea un reto.

Poner en práctica políticas en toda la red y responder al amplio rango de eventos que pueden ocurrir, (como pueden ser cambios en el tráfico o intrusiones) resulta una terea realmente complicada, debido a que, se pueden tener que configurar miles de dispositivos y aplicar diversos mecanismos, mediante un conjunto de comandos de configuración de bajo nivel muy limitado. Además, la complejidad de las redes de hoy en día, hace que sea muy complicado aplicar un conjunto consistente de acceso, seguridad, de calidad de servicio (QoS) y otras políticas, al cada vez mayor, número de usuarios móviles.

Otra de las limitaciones de las redes actuales la generan los propios ISP (Internet Service Provider) ya que debido a la falta de estandarización se limita la capacidad de los operadores de red para adaptar la red a sus entornos particulares de una manera interoperable.

Este desajuste entre las necesidades del mercado y las capacidades de la red ha llevado a la industria a un punto de inflexión. En respuesta, la industria y la comunidad científica ha desarrollado las arquitecturas de red definida por software (SDN) y sus estándares asociados. SDN se basa en la separación entre los planos de control y datos en los equipos de red haciendo que la gestión, evolución y funcionamiento de los mismos se simplifique añadiendo aproximaciones software centralizadas

Debido a la evolución que se espera de las redes hacia esta arquitectura y a la importancia de los sistemas de monitorización para la gestión y mantenimiento de las redes, en este trabajo se propone un sistema de monitorización totalmente integrado en entornos SDN.

1.2 Objetivos

El objetivo de este trabajo es implementar un sistema de monitorización para redes SDN. Ese sistema permitirá aplicar políticas de calidad de servicio y seguridad aplicando reglas instalables en los equipos de red a través del protocolo OpenFlow. El sistema analizará todo el tráfico de una red y generará un conjunto de reglas y políticas que permitirán gestionar el tráfico de manera autónoma.

Para alcanzar esta meta se han definido los siguientes objetivos:

- Documentación y asimilación de conceptos relacionados con la SDNs y OpenFlow.
- Análisis y configuración del entorno de trabajo basado en el emulador de red Mininet.
- Implementación y pruebas del sistema de monitorización propuesto.

1.3 Organización de la memoria

Este documento se organiza en cinco capítulos. El primer capítulo sirve de Introducción, mostrando la motivación y objetivos de este trabajo de fin de grado, junto con la organización del documento. En el segundo capítulo, Estado del Arte, se muestra un estudio de las redes definidas por software, del protocolo OpenFlow, además de un análisis de tecnologías similares encontradas en la literatura. En el tercer capítulo, Diseño y Desarrollo, se muestra cómo se ha desarrollado el sistema de monitorización. En el cuarto capítulo, Pruebas y Resultados, se presentan las pruebas y resultados relevantes obtenidos para validar el funcionamiento del sistema así como su rendimiento. Por último se presenta el capítulo 5 con las Conclusiones más relevantes de este trabajo así como las mejoras que podrían añadirse en un futuro.

2 Estado del arte

2.1 Red definida por Software (SDN)

SDN [3] es un nuevo paradigma dentro de las redes de comunicaciones, que se basa en la separación entre los planos de control y datos en los equipos de red y es directamente programable.

Esta migración del control, desde los dispositivos individuales de red, a dispositivos computacionalmente accesibles, permite que la infraestructura subyacente sea abstraída para que aplicaciones y servicios de red puedan tratar a la red como una entidad lógica o virtual.

La Figura 2-1 [4] representa una vista lógica de la arquitectura SDN, donde la inteligencia de la red está en el controlador. El controlador de una SDN es una entidad lógicamente centralizada, esto quiere decir, que puede consistir en múltiples instancias físicas o virtuales, pero se comporta como un único componente. Una red o red virtual tiene un controlador que mantiene un estado global de esa red o fragmento de red particular, de este modo las empresas y operadores ganan control sobre toda la red desde un único punto lógico, lo que simplifica enormemente el diseño y operación de la red. SDN también simplifica los dispositivos de red en sí, ya que no necesitan entender y procesar miles de protocolos sino simplemente aceptar las instrucciones del controlador SDN.

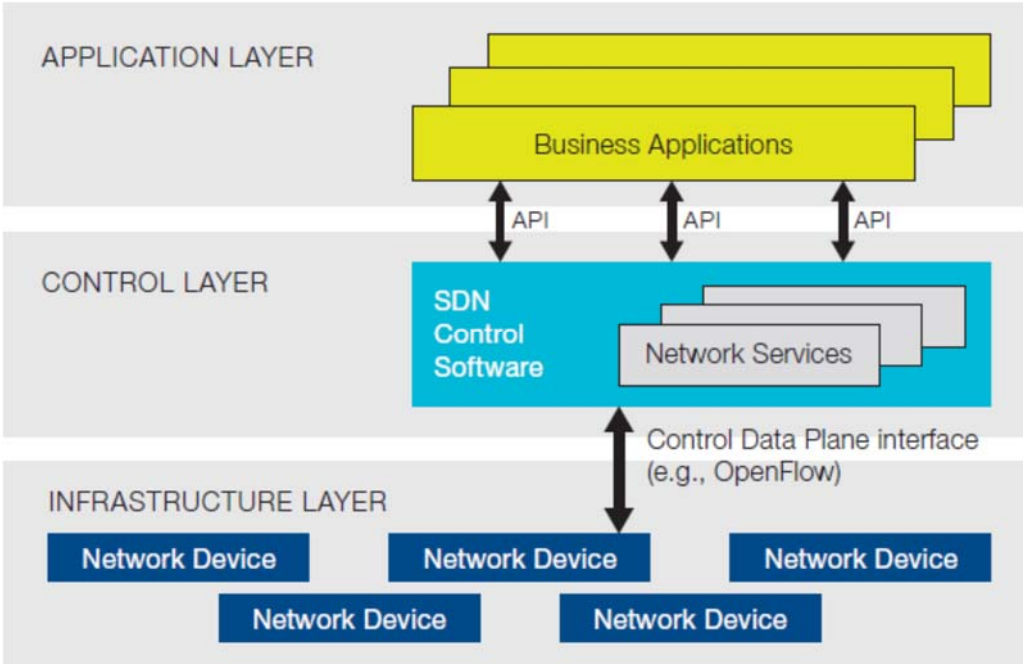


Figura 2-1: Arquitectura SDN

Uno de los puntos más importantes de SDN, es la "programabilidad" de la red, como se afirma en [5], que representa la habilidad de tratar a la red como una única entidad programable en vez de una acumulación de dispositivos que tienen que ser configurados

individualmente. Además, gracias a la inteligencia centralizada que proporciona el controlador SDN, es posible alterar el comportamiento de la red en tiempo real y desplegar nuevas aplicaciones y servicios de red en cuestión de horas o días, en lugar de las semanas o meses que se necesitan en la actualidad.

Centralizando el estado de la red en el plano de control, SDN ofrece a los administradores de red la flexibilidad de configurar, gestionar, asegurar, y optimizar los recursos de red de manear dinámica y automática a través de programas SDN. Además, los administradores de red pueden escribir estos programas ellos mismos, de manera que no hay que esperar a que ciertas características o funcionalidades vengan integradas por el fabricante.

Además de la abstracción de la red, la arquitectura SDN soporta una serie de APIs que hacen posible implementar servicios de red comunes, incluyendo rutado, envío de tráfico multicast, seguridad y control de acceso, gestión del ancho de banda, ingeniería de tráfico, calidad de servicio, optimización de recursos físicos, gestión de energía, y, en definitiva, cualquier tipo de política de gestión personalizada que sirva para conseguir los objetivos de negocio.

2.2 OpenFlow

Openflow [6,7] es la primera interfaz de comunicación estándar entre los planos de control y datos de una arquitectura SDN. OpenFlow permite el acceso directo y una manipulación del plano de datos de dispositivos de red como switches y routers, tanto físicos como virtuales.

Openflow utiliza el concepto de flujo para identificar el tráfico de red basándose en reglas de correspondencia (*matching*) predefinidas, que pueden se programadas de manera estática o dinámica por el controlador SDN. Para ello el controlador consta de una tabla de flujos o reglas, con una serie de datos sobre el paquete para realizar la coincidencia, una serie de contadores para obtener estadísticas de los flujos activos y ninguna o varias acciones que realizar. Los datos para realizar la coincidencia del paquete para la versión 1.0.0 de OpenFlow, como vemos en [8], puede ser cualquier combinación de los siguientes campos:

- Puerto de entrada al switch
- Direcciones Ethernet origen y destino
- Ethertype
- VLAN ID
- Prioridad VLAN
- Direcciones IPv4 origen y destino
- Protocolo IPv4
- ToS de IPv4
- Puertos TCP o UDP origen y destino.

Dado que OpenFlow permite a la red ser programada en una base de flujos, una arquitectura SDN basada en OpenFlow proporciona un control muy granular, permitiendo a la red responder en tiempo real a cambios en los niveles de aplicación, usuario y sesión. Las redes actuales basadas en rutado IP no proporcionan este nivel de control, debido a que todos los flujos entre dos extremos deben seguir el mismo camino a través de la red, sin tener en cuenta los diferentes requisitos del tráfico. Dependiendo de las reglas que se instalen en el switch

por el controlador, un switch OpenFlow se puede comportar como un router, un switch, un firewall, un traductor de direcciones de red o cualquier aplicación de red similar.

Las SDN basadas en OpenFlow se pueden desplegar en las redes existentes, tanto físicas como virtuales siempre que los dispositivos de red lo soporten. Los dispositivos de red pueden soportar rutado basado en OpenFlow tanto como el tradicional, lo que hace muy sencillo para las empresas y proveedores la progresiva introducción de las tecnologías SDN basadas en OpenFlow, incluso en entornos de red con varios fabricantes.

OpenFlow presenta las siguientes ventajas:

- **Control centralizado en entornos con distintos fabricantes:** el software de control SDN puede controlar cualquier dispositivo de red compatible con OpenFlow de cualquier fabricante, incluyendo switches, routers, y switches virtuales. En vez de tener que gestionar grupos de dispositivos de distintos fabricantes, además se pueden utilizar las herramientas de orquestación y gestión SDN para desplegar, configurar y actualizar de manera rápida dispositivos a lo largo de toda la red.
- **Reducción de la complejidad gracias a la automatización:** SDN basado en OpenFlow ofrece un entorno de gestión y automatización de red flexible, que hace posible desarrollar herramientas para automatizar muchas tareas de gestión, que se hacen a mano actualmente. Estas herramientas de automatización reducirán la sobrecarga operacional y disminuirá la inestabilidad de la red introducida por errores del operador. Además, con SDN, las aplicaciones basadas en la nube se pueden gestionar a través de sistemas de orquestación y aprovisionamiento inteligente, reduciendo aún más los gastos operativos y aumentando la agilidad del negocio.
- **Incremento de la innovación:** la adopción de SDN acelera la innovación empresarial permitiendo que los operadores de red configuren la red en tiempo real para satisfacer las necesidades específicas de negocio y necesidades de los usuarios a medida que surjan. Virtualizando la infraestructura de red y abstrayéndola de los servicios de red individuales, por ejemplo OpenFlow proporciona a los operadores de red y potencialmente incluso a los usuarios, la habilidad de adaptar el comportamiento de la red e introducir nuevos servicios y capacidades de red en cuestión de horas.
- **Aumento de la fiabilidad y la seguridad de la red:** SDN hace posible para los operadores de red definir configuraciones y políticas que luego son trasladadas a la infraestructura a través de OpenFlow. Una arquitectura SDN basada en OpenFlow elimina la necesidad de configurar individualmente los dispositivos de red cada vez que un dispositivo final, servicio o aplicación es añadida o movida, o hay un cambio en la política de la red, lo que reduce la probabilidad de fallos de la red debido a configuración, o a políticas inconsistentes.

  Gracias a que los controladores SDN proporcionan control y una visión completa de la red, se pueden aplicar políticas de control de acceso, ingeniería de tráfico, calidad de servicio y seguridad de manera consistente a lo largo de las infraestructuras de red tanto cableadas como inalámbricas. Las empresas y proveedores se benefician reduciendo los gastos de operación, con capacidades más dinámicas de configuración, menos errores y una configuración consistente y refuerzo de las políticas.

- **Control de red más granular:** el modelo de control basado en flujos de OpenFlow, permite aplicar políticas a un nivel muy granular, incluyendo los niveles de sesión, usuario, dispositivo y aplicación, de una manera muy abstraída y automatizada. Este control permite a los operadores de la nube soportar múltiples clientes al mismo tiempo asegurando el aislamiento del tráfico, la seguridad y una gestión elástica de los recursos cuando los clientes comparten la misma infraestructura.
- **Mejora de la experiencia de usuario:** Centralizando el control de la red y teniendo información de estado disponible para las aplicaciones de alto nivel, una infraestructura SDN puede adaptarse mejor a las necesidades dinámicas de cada usuario. Por ejemplo, un proveedor puede introducir un servicio de video que ofrece a los subscriptores Premium la mejor resolución posible de una manera automática y transparente. En la actualidad, los usuarios deben seleccionar explícitamente la resolución, que la red puede o no ser capaz de soportar, causando retardos e interrupciones que degradan la experiencia de usuario. Con SDN basado en OpenFlow, las aplicaciones de video serán capaces de detectar el ancho de banda disponible en la red en tiempo real y en función del mismo, automáticamente adaptar la resolución del video.

Controladores OpenFlow

Las plataformas de controlador OpenFlow permiten desarrollar aplicaciones de red, obviando la complejidad y heterogeneidad de la red subyacente. En un controlador OpenFlow se puede hacer referencia a dos interfaces: la denominada Southbound API hace referencia a la interfaz y protocolo entre los switches programables (Switches SDN) y el software del controlador y la denominada Northbound API determina como expresar tareas operacionales y políticas de red y como traducirlas a una forma que el controlador pueda entender.

En los últimos años, se han desarrollado varias plataformas de controlador escritas en distintos lenguajes, como se explica en [9]. La primera que se creo fue NOX que soporta Python y C++, también está POX, simplemente para Python (desarrollada por el mismo equipo que NOX) que permite un desarrollo rápido y sencillo. También escrita en Python encontramos Ryu. Escritas en Java están Beacon y Floodlight que también destacan por la simplicidad a la hora del desarrollo de aplicaciones de red. También existe una opción escrita en Ruby llamada Trema.

Todos los controladores mencionados anteriormente, son de código abierto. Además de los distintos lenguajes que presentan también cabe destacar ciertas prestaciones diferenciadoras. Por ejemplo, POX permite debugging, virtualización de red, un diseño del controlador mediante modelos de programación. Por otro lado, Ryu es un controlador SDN basado en componentes predefinidos, que pueden ser modificados extendidos y combinados para crear una aplicación personalizada de controlador. Trema, también destaca ya que entre sus objetivos de diseño están la facilidad en la escritura de código y el rendimiento. El lenguaje basado en scripts “Ruby” se emplea para aumentar la productividad y se emplea un compilador de lenguaje C para aumentar el rendimiento.

2.3 Sistemas de monitorización

Las redes definidas por software tienen como objetivo simplificar y mejorar el control y la gestión de red, al mismo tiempo que se simplifica la implementación de nuevas aplicaciones. Una clase de aplicaciones que se pueden beneficiar especialmente de las características de las SDN y de OpenFlow son las aplicaciones de monitorización de red, como se puede ver en [10].

Un sistema de monitorización de red obtiene datos de los paquetes que circulan por la red, para analizarlos y poder extraer conclusiones. Por ejemplo se puede detectar un mal funcionamiento de la red como puede ser la congestión de la red o también algún tipo de ataque. Además frente a la monitorización clásica, en los entornos SDN podemos actuar de manera dinámica ante los problemas aplicando políticas de calidad de servicio o seguridad. En las redes tradicionales la monitorización carecía de proactividad y necesitaba de gestores de red que analizasen los resultados de la monitorización y aplicasen de manera manual la política de seguridad o calidad de servicio que correspondiese.

Recientemente han surgido en la literatura varios sistemas de monitorización basados en OpenFlow, en los siguientes apartados se explican los más relevantes.

2.3.1 PayLess

En [11] se presenta PayLess, un entorno de recolección de estadísticas de red. Payless está desarrollado sobre múltiples plataformas de controlador OpenFlow y proporciona una API REST de alto nivel. La API REST que proporciona PayLess permite a los administradores de red desarrollar aplicaciones de monitorización. El entorno de monitorización se encarga de traducir los requerimientos de alto nivel de las aplicaciones, ocultando detalles de la recolección de estadísticas, almacenamiento y gestión. De esta manera. Las aplicaciones de monitorización de red, implementadas sobre este entorno permanecerán aisladas de los detalles de bajo nivel. Un controlador puede solicitar a un switch que recolecte estadísticas de los flujos activos para luego hacer uso de estas estadísticas. Payless consta de una serie de componentes para determinar a partir de los requerimientos de las aplicaciones de alto nivel cuando y a que switches solicitar las estadísticas. En la Figura 2-2 puede observarse el diagrama de funcionamiento de PayLess.

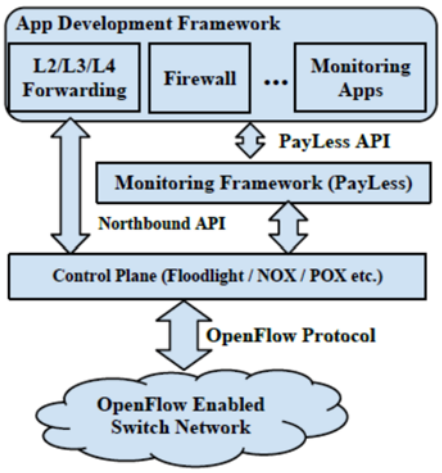


Figura 2-2: Diagrama PayLess

2.3.2 Lightweight

En [12] se presenta Lightweight, un mecanismo de detección de ataques distribuidos de denegación de servicio (DDoS). Lightweight se basa en el análisis de las características del tráfico a nivel de flujo. Para obtener dichas estadísticas Lightweight, a través de controlador NOX, monitoriza los switches registrados durante intervalos de tiempo predeterminados. Durante esos intervalos se extraen las características de interés de las entradas de la tabla flujos de todos los switches. Cada muestra luego es clasificada por un sistema inteligente de detección de ataques, que determinará si esta información corresponde a tráfico normal o a un ataque. En la Figura 2-3 se muestra a arquitectura de Lightweigth.

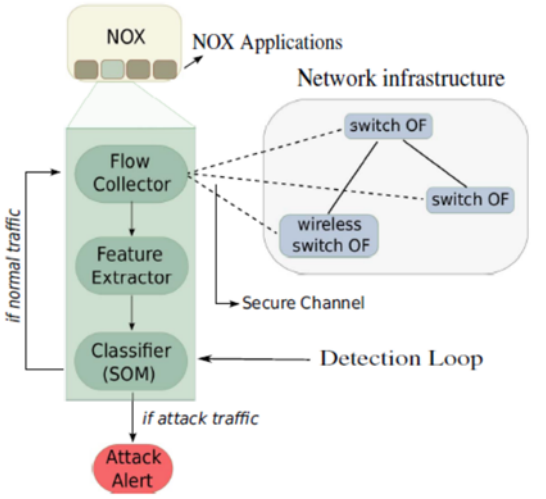


Figura 2-3: Diagrama Lightweight

2.3.3 CloudWatcher

CloudWatcher [13], proporciona servicios de monitorización para redes Cloud de gran tamaño y dinámicas. Este sistema consiste añade en puntos específicos de la red dispositivos de seguridad y desvía los paquetes de red mediante varios algoritmos, para que todo el tráfico que se quiera analizar sea inspeccionado por alguno de los dispositivos de seguridad. Los dispositivos de seguridad están basados en controladores OpenFlow. En este sistema se ha desarrollado una política basada en scripts para que el administrador de red pueda indicar al controlador que políticas de seguridad desea aplicar a la red. El diagrama del sistema se muestra en la Figura 2-4.

.

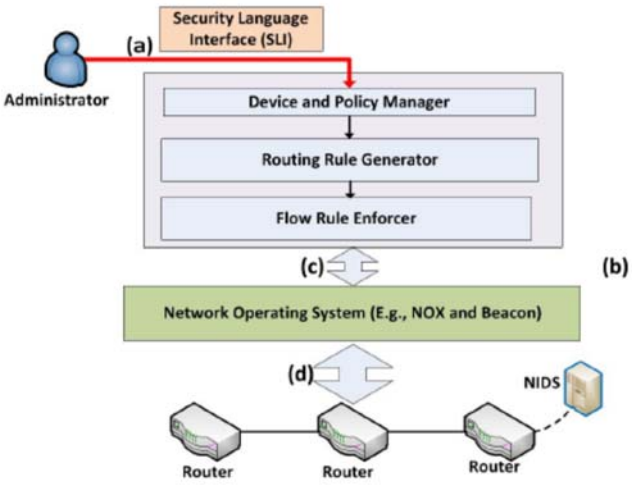


Figura 2-4: Diagrama CloudWatcher

2.3.4 FRESCO

FRESCO [14] es un entorno de desarrollo de aplicaciones de seguridad en OpenFlow. Este entorno de desarrollo presenta una API de scripts que permite a los administradores de red, crear un sistema de monitorización para seguridad y detección de amenazas en forma de librerías modulares. Estas librerías modulares representan la unidad de procesado elemental de FRESCO, y pueden ser compartidas y unidas para generar complejas aplicaciones de defensa de red como firewalls, scan detectors, attack deflectors o sistemas de detección de intrusión (IDS). En la Figura 2-5 la arquitectura de diseño de FRESCO.

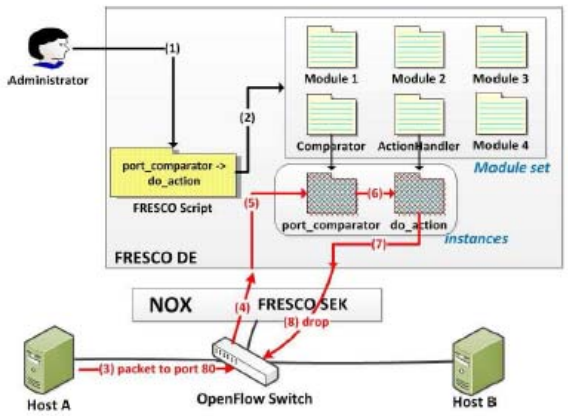


Figura 2-5: Diagrama FRESCO

Analizando estos trabajos relacionados con la monitorización en redes definidas por software, podríamos dividir los trabajos estudiados en trabajos de detección o análisis que permiten conocer el estado de la red mediante gráficas o tablas y que avisan de un cierto ataque; como pueden ser Payless o Lightweight. Por otro lado tenemos sistemas como CloudWatcher y FRESCO que pretenden ser una herramienta para que los administradores de red desarrollen sus propios sistemas de seguridad o políticas de tráfico, de manera más sencilla, sin tener que encargarse de detalles de bajo nivel del protocolo OpenFlow.

Nuestro sistema combina características de ambos tipos de sistemas de monitorización. Por un lado analiza todo el tráfico de red detectando ciertas anomalías en la red y por otro aplica políticas en la red para la normalización de dicho tráfico. La gran diferencia de nuestro sistema es la automatización de la creación de las políticas que necesita la red

También cabe recalcar que, todos los sistemas descritos anteriormente utilizan los datos de los flujos activos recogidos por los switches, para obtener sus estadísticas o determinar la existencia de algún ataque, esa información de los switches se obtiene a partir del controlador lo que puede suponer una sobrecarga del mismo. Además obteniendo la información a través del controlador no se tiene acceso a todo el tráfico de manera que las estimaciones realizadas por los sistemas pueden ser menos precisas.

Otro inconveniente que resulta de la utilización del controlador para obtener la información del tráfico, surge cuando hay una anomalía en un tráfico para el que el controlador ya ha instalado una regla ya que este tráfico no se inspecciona hasta que la misma expire.

En nuestro sistema se ha desarrollado un elemento externo para la obtención de los datos necesarios del tráfico mientras que el controlador simplemente se encarga de actualizar la tabla de flujos del switch. De esta manera evitamos los problemas que pueden surgir por la utilización del controlador para la obtención de la información del tráfico de red a analizar.

3 Diseño y desarrollo

En este proyecto se propone la implementación de un sistema de monitorización integrado en entornos SDN. Este sistema de monitorización permitirá la aplicación de políticas de calidad de servicio o seguridad a través de reglas instalables en los equipamientos de red usando el protocolo OpenFlow. El sistema analizará todo el tráfico de una red y generará un conjunto de reglas y políticas que permitirán gestionar el tráfico de manera inteligente.
El sistema de monitorización que se va a desarrollar consta de dos elementos principales:

- El primer elemento es, el controlador característico de las SDN que se encarga de controlar la red modificando la tabla de flujos de los switches OpenFlow. Si un switch OpenFlow recibe un paquete y no encuentra una coincidencia en su tabla de reglas de flujos lo envía al controlador para que tome una decisión de cómo tratar el paquete.
- La segunda parte del sistema es un analizador de tráfico que trata con todo el tráfico que circula por la red y genera un fichero con las reglas que serán utilizadas por el controlador para actualizar la tabla de flujos del switch. Dicho programa se colocará en una interfaz de red del switch a la que se copia todo el tráfico que circula por el mismo, lo que se denomina “port mirroring”. El programa analiza pues todo el tráfico de la red y de acuerdo a ciertos parámetros de los flujos como los bits/s o el número de paquetes SYN, establece unas acciones u otras para cada flujo en particular. La arquitectura del sistema de monitorización se muestra en la Figura 3-1.

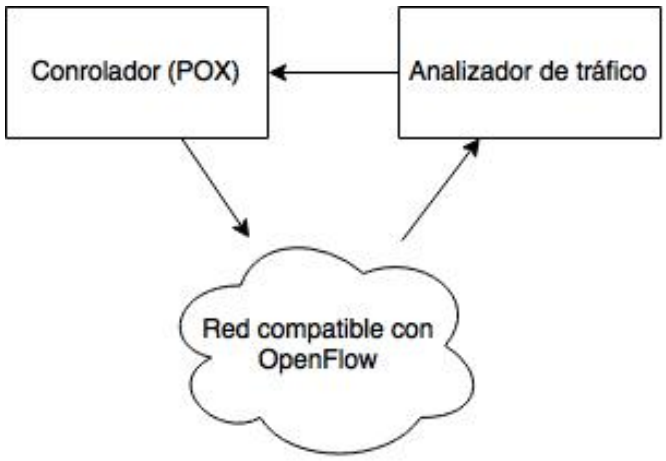


Figura 3-1: Diagrama del sistema de monitorización diseñado

3.1 Controlador OpenFlow

En las SDN cada elemento de red, típicamente un switch OpenFlow, simplemente envía los paquetes que recibe de acuerdo a su tabla de flujos. Si un paquete no tiene ninguna coincidencia en su tabla este se envía al controlador, que de acuerdo a la programación indicará como tratar el paquete añadiendo una nueva entrada en la tabla del switch o elemento de red.

En este trabajo se utilizará una de las plataformas de controlador OpenFlow, para desarrollar las funcionalidades deseadas en el controlador del sistema de monitorización, concretamente se utilizará POX. La elección de esta plataforma se debe a la facilidad de uso así como a su gran popularidad. POX permite generar prototipos de sistemas y servicios rápidamente para evaluar la viabilidad de las soluciones lo cual encaja perfectamente con los requisitos de este trabajo de fin de grado.

Para la programación del funcionamiento del controlador se ha partido de un script ya incluido en la librería de POX que provee la funcionalidad básica de un switch. El funcionamiento es el siguiente,

- Cuando llega un paquete se guarda la dirección MAC origen y el puerto por el que ha llegado en una tabla con que relaciona ambos parámetros. En el futuro si llega un paquete con dicha dirección MAC como destino se sabrá por qué puerto debe enviarse.
- Si la dirección destino es multicast o no se encentra en la tabla que relaciona direcciones MAC con puertos, se hace envía el paquete por todos los puertos haciendo flooding como en un Hub clásico.
- Si el puerto de salida es el mismo que el de entrada se descarta el paquete y paquetes similares por un tiempo para evitar bucles.
- Si en la tabla que relaciona direcciones MAC y puertos se encuentra el puerto por el que se debe enviar el paquete se añade una entrada a la tabla de flujos del switch OpenFlow y se envía el paquete.

A partir de este funcionamiento básico del switch, se ha modificado el script para que rellene la tabla de flujos OpenFlow de acuerdo a un fichero de texto con las reglas (políticas) que requiere la red, como por ejemplo tirar (drop) el tráfico de un determinado flujo, modificar la prioridad o tipo de servicio (TOS), añadir una cabecera VLAN etc. La modificación más concretamente se ha basado en la creación de un hilo de proceso que lee y guarda en una tabla el fichero de reglas el cual tiene el formato mostrado en la Tabla 3-1 por cada línea.

IP origen	IP destino	Puerto origen	Puerto destino	Protocolo	Acción

Tabla 3-1: Formato del fichero de reglas

Para cada paquete, después de comprobar si el puerto de salida es el mismo que el de entrada, se comprueba si el paquete coincide con alguna de las reglas contenidas en el fichero. En ese caso en vez de enviarlo por el puerto que corresponde según la tabla de MAC y puertos, se aplica la acción indicada en el campo acción del fichero. Dichas acciones pueden ser:

- FlOOD: se envían todos los paquetes entrantes por cada interfaz de salida, excepto por la que se ha recibido. A continuación se muestra un ejemplo de una regla con esta acción:

10.0.0.1 10.0.0.2 2000 2001 TCP FLOOD

- DROP: se descarta el paquete y no se envía por ninguna interfaz. A continuación se muestra un ejemplo de una regla con esta acción:

10.0.0.1 10.0.0.2 2000 2001 TCP DROP

- SET_TOS: al paquete se le cambia el tipo de servicio (TOS) en la cabecera IP por el valor especificado en el fichero, como veremos a continuación en un ejemplo.

10.0.0.1 10.0.0.2 2000 2001 TCP SET_TOS 32

- PUSH_VLAN: se añade un cabecera VLAN con el ID especificado en el fichero. En el fichero una regla con esta acción tiene el siguiente aspecto

10.0.0.1 10.0.0.2 2000 2001 TCP PUSH_VLAN 20

- OUTPUT: se envía el paquete por la interfaz del switch especificada en el fichero. Una regla con esta acción tiene el siguiente aspecto:

10.0.0.1 10.0.0.2 2000 2001 TCP OUPUT 3

Si el paquete que ha llegado no coincide con ninguna de las reglas del fichero, se envía normalmente usando la información contenida en la tabla de MACs y puertos. Para poder aplicar políticas sobre el tráfico de manera más flexible, a la hora de comparar los campos de un paquete entrante con los del fichero, se admite la utilización de un comodín (“*”) en el fichero que permite por ejemplo aplicar una acción a todo el tráfico con destino 10.0.0.2 como vemos en el siguiente ejemplo:

** 10.0.0.2 * * * DROP*

Gracias a POX la implementación de estas acciones se realiza de manera sencilla, debido a una serie de funciones y macros. Para hacer flooding basta con indicar que la acción que se va a añadir en la tabla de flujo del switch, es enviar el tráfico por todas las interfaces menos por la de entrada. Para ello POX cuenta con la acción of.OFPP_FLOOD. En el siguiente ejemplo se muestra cómo se puede añadir una acción de flood.

```
msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
```

POX proporciona objetos que representan los mensajes OpenFlow que se enviarán al switch o equipamiento de red. En este caso msg es un objeto de dicha clase.
Para descartar los paquetes (drop), se añade una entrada en la tabla de flujos del switch sin especificar ninguna acción. Para añadir una entrada en el switch con la acción OUTPUT

basta con especificar que la acción es enviar por la interfaz especificada en el fichero. En el siguiente ejemplo se muestra cómo se puede añadir una acción para sacar todo el tráfico de ese flujo por un interfaz determinada contenida en una variable llamada “port_file”:

```
msg.actions.append(of.ofp_action_output(port = port_file))
```

Para añadir una entrada en el switch con la acción SET_TOS, realmente se añaden varias acciones: primero se modifica la cabecera IP con el nuevo TOS y luego se envía por la interfaz adecuada para llegar a su destino. En el siguiente ejemplo se muestra cómo se añaden las dos acciones de manera secuencial.

```
nw_action = of.ofp_action_nw_tos()
nw_action.nw_tos = tos #valor obtenido del fichero
msg.actions.append(nw_action)
msg.actions.append(of.ofp_action_output(port = port))
```

Añadir una entrada en el switch con la acción PUSH_VLAN, se realiza de manera muy similar a la acción SET_TOS.

```
vlan_action = of.ofp_action_vlan_vid()
vlan_action.vlan_vid = vlan_id #valor obtenido del fichero
msg.actions.append(vlan_action)
msg.actions.append(of.ofp_action_output(port = port))
```

Como función adicional, se ha modificado el controlador para que copie todo el tráfico a una interfaz conocida (port mirroring) donde se encontrará el programa analizador de tráfico que genera las reglas.

3.2 Analizador de tráfico

Este programa se ha realizado en lenguaje C y utiliza la librería PCAP de UNIX, que permite la captura y análisis de paquetes de red. De cada paquete que se recibe se obtiene, primero la versión IP ya que solo nos interesan los paquetes con IP versión 4 y así descartar los paquetes con versión 6. Además únicamente se toman en cuenta los paquetes UDP y TCP ya que son los protocolos de nivel 4 más comunes. Para los paquetes que cumplen los anteriores criterios se guarda en una estructura de datos los siguientes campos:

- IP origen
- IP destino
- TOS
- Protocolo contenido en IP
- Puerto origen
- Puerto destino
- Número de bits del paquete
- Bandera SYN (si es TCP)

Una vez tenemos los datos que nos interesan del paquete en una estructura, se añaden a una tabla *hash* de direccionamiento abierto. Dicha tabla ha sido implementada desde cero para

adaptarla a las necesidades de este proyecto. Como índice se calcula un hash a partir de las IPs y puertos (origen/destino), y el protocolo; de manera que se agregue la información a nivel de flujo.

La tabla se irá actualizando con cada nuevo paquete de la siguiente manera:

1. Se calcula el valor hash de un paquete y se guarda como “hash inicial”.
2. Se comprueba si la posición resultante está vacía y si es así se guarda el paquete en esa posición, guardando en un campo el número de bits totales del flujo y el número de SYNs totales del flujo.
3. Si la posición resultante está ocupada se compara el contenido de la posición ocupada con el del nuevo paquete ya que dos paquetes de distintos flujos pueden generar el mismo hash debido a colisiones. Si se trata del mismo flujo se actualizan solo los campos de número de bits totales y número de SYNs totales.
4. Si no se trata el mismo flujo se recalcula el hash, para obtener uno nuevo y se vuelven a realizar todos los pasos anteriores, hasta guardar el paquete en una posición de la tabla.
5. En caso de que el valor del hash recalculado coincida con el valor de “hash inicial”, esto indicará que la tabla se ha llenado, en ese caso se ampliará la tabla y se volverá al paso 1.

La función hash utilizada en este proyecto consiste en la siguiente fórmula:

$$hash = (IP\_src + IP\_dst + Port\_src + Port\_dst + Proto)\%tam\_tabla$$

Y en caso de que haya que recalcular el hash se realiza siguiendo una secuencia de sondeo cuadrática similar a la siguiente:

```
perturb = y;
hash = (5*hash)+1+perturb;
perturb >> 5;
hash =hash % tam_tabla;
```

De manera paralela se ejecuta un hilo de proceso que recorre esta tabla hash y genera las reglas que serán escritas a un fichero llamado file.rules. Solo se han generado tres tipos de reglas como ejemplo ilustrativo de la clase de análisis que se podrían aplicar en este sistema. Los tres tipos de reglas son las siguientes:

1. Número de paquetes TCP SYN que recibe una dirección IP destino desde una dirección IP origen. Si se sobrepasa un umbral (configurable) de 4 SYN/seg se descarta el tráfico que vaya desde el origen al destino a través de los mismos puertos. Esto protege contra un ataque de denegación de servicio (DoS) basado en SYNs.
2. Tasa en bits/s del flujo. Si se supera el 50% de la capacidad del enlace, se descarta el tráfico perteneciente a dicho flujo. Esto evita que un solo flujo ocupe la mayor parte de los recursos del sistema.
3. Tasa de bits/s del flujo. Si se supera el 20% de la capacidad del enlace. En este caso el tráfico se marcará con prioridad baja (TOS 8). Esto hace que los flujos que requieren una tasa de tráfico alta sean tratados con menos prioridad.

Para evitar picos instantáneos en el tráfico, se promedian las medidas cada 10 segundos. Este umbral es configurable manualmente para adaptarse a las condiciones de la red a monitorizar. Todos estos umbrales están accesibles en el código para poder modificarlos en función de la red. Todo el código desarrollado está libre y accesible en el siguiente repositorio: https://bitbucket.org/sdnmon/sdnmon/src

4 Pruebas y resultados

En este apartado se describen las pruebas realizadas para validar el comportamiento y rendimiento del sistema. Concretamente se realizarán tres pruebas separadas para generar cada tipo de regla por separado y por ultimo una evaluación del rendimiento global del programa.

Las pruebas de validación consistirán en la generación del entorno adecuado para que el sistema se comporte de la manera esperada. Lo que se quiere conseguir es que el programa genere la regla o reglas adecuadas en cada momento y que el controlador indique a los switchs qué hacer con los flujos de tráfico para los que se ha generado una regla.

Las pruebas de rendimiento consistirán en ver la cantidad de paquetes por segundo y la cantidad de bits por segundo que el sistema puede procesar en su operación normal.

4.1 Entorno de pruebas

Las pruebas para validar el sistema se realizaran sobre una topología de red virtual, emulada en Mininet [15], en una máquina virtual que funciona sobre Kernel Virtual Machine (KVM) y que cuenta con una CPU virtual de 64 bits con 8 cores a 2GHZ y con 32 GB de RAM. La topología de prueba consta de siete hosts, tres switches y dos controladores, como vemos en la Figura 4-1Uno de los siete hosts, concretamente h7, es donde se encontrará el programa que genera el fichero file.rules y a donde se copiará el tráfico. Los switches s1 y s3, tendrán como controlador un controlador de switch estándar mientras que el switch s2 tendrá el controlador desarrollado en este trabajo; para esta red de reducido tamaño se ha escogido un punto crítico por donde pasa todo el tráfico para establecer el sistema de monitorización SDN formado por los elementos c0 (controlador) y h7 (analizador de tráfico). A la hora de crear los enlaces entre los host se ha especificado un ancho de banda de 100 Mb/s.

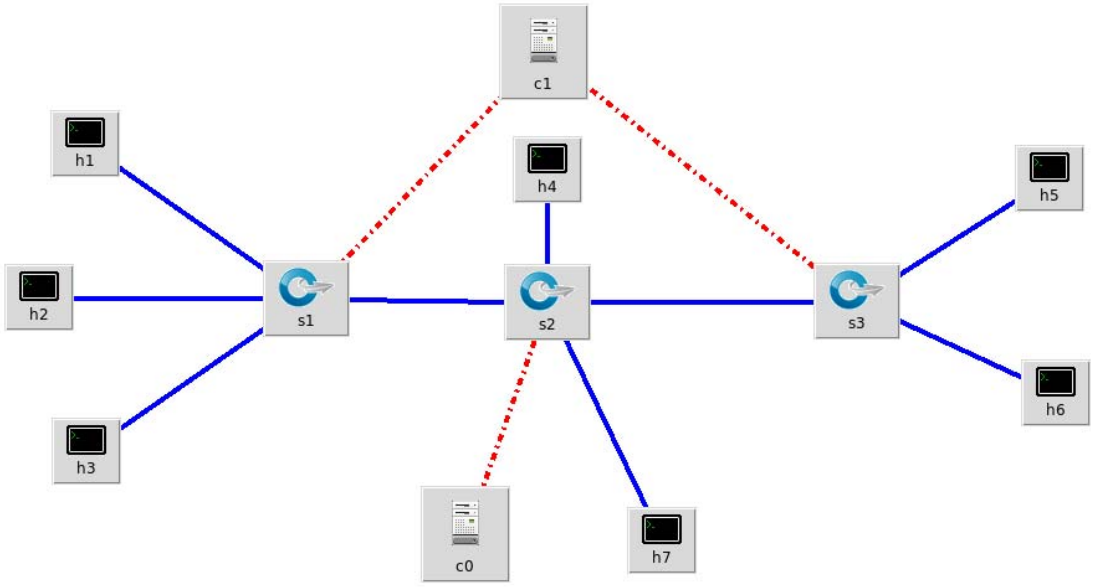


Figura 4-1: Topología en Mininet

Al crear la topología se han especificado las IPs de los distintos host según muestra la Tabla 4-1.

Host	IP	host	IP
h1	192.168.1.26	h4	192.168.1.1
h2	192.168.1.21	h5	192.168.1.2
h3	192.168.1.24	h6	192.168.1.3

Tabla 4-1: Asignación de IPs

También se ha definido el tipo de controlador, en nuestro caso puesto que usamos un controlador remoto se ha seleccionado la opción de "Controller Remote".

Una vez está definida la topología la levantamos o iniciamos. Si después de esto intentamos enviar paquetes entre los hosts no será posible ya que los controladores remotos no están iniciados. En una topología se pueden añadir varios controladores pero no se pueden iniciar dos controladores con la misma IP en el mismo puerto, así que se han definido en el puerto por defecto 6633 y en el puerto 6634. Utilizamos POX para iniciar los controladores de la siguiente manera:

./pox .py myl2_learning
. /pox .py openflow.of_01 –port=6634 l2_learning

Donde myl2_learning es el módulo controlador desarrollado en el ámbito de este trabajo y l2_learning es el módulo controlador estándar que proporciona funcionalidades de switch cásico. Después se arranca el analizador de tráfico en el host h7quedando el sistema completo para la realización de las pruebas.

4.2 Pruebas de validación

En esta topología se generará tráfico entre los hosts para forzar la generación de los diferentes tipos de reglas en el fichero *file.rules*. Dichas reglas servirán para hacer frente a distintas situaciones que pueden alterar el correcto funcionamiento de la red y así corregirlo.

4.2.1 Ataque por inundación de SYN

Un ataque de inundación de SYN (SYN flooding) es un tipo de ataque de denegación de servicio (DoS). En este ataque se envía un gran número de segmentos SYN TCP, sin completar el handshake en tres fases de TCP. Cuando la tasa de paquetes SYN generados es muy alta el servidor pasa más tiempo respondiendo a paquetes SYN (y reservando recursos para las conexiones) que atendiendo peticiones legítimas lo cual provoca una caída en el servicio.

En esta prueba queremos mostrar que el programa es capaz de detectar un ataque SYN flooding desde una IP y puerto origen a una IP y puerto destino con la consecuente actualización del fichero file.rules para que ese flujo de tráfico se descarte. Además se quiere

comprobar que una vez la regla está en el fichero file.rules el controlador actúa de manera consecuente a esta regla.

Para ello se va a generar tráfico entre los hosts h1 y h4. Para realizar el SYN flooding se utilizará hping3 [16] para enviar tráfico TCP con el bit SYN activado del siguiente modo:

hping3 –S --fast –s <Port_src> -k –p <Port_dst> <IP_dst>

Como resultado esperamos que si la tasa de SYN/seg desde h1 a h4 supera el umbral de 4 SYN/seg especificado en el programa para considerar que la situación planteada se está produciendo, se genere la regla con el formato correcto y que el switch comience a descartar los paquetes que coincidan con los campos de IP origen y destino, puertos origen y destino y protocolo de la regla.

El resultado real de esta prueba es el siguiente:

Display

Display filter: ip.dst == 192.168.1.1

Ignored packets: 0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	170	84	49,412%	0	0,000%
Between first and last packet	8,276 sec	8,276 sec			
Avg. packets/sec	20,540	10,150			
Avg. packet size	53,859 bytes	54,000 bytes			
Bytes	9156	4536	49,541%	0	0.000%
Avg. bytes/sec	1106,287	548,074			
Avg. MBit/sec	0,009	0,004			

Figura 4-2: Tráfico ataque por inundación SYN

Como vemos en la Figura 4-2, hasta generarse la regla se ha alcanzado una tasa de 10.150 SYN/seg. Puesto que el umbral para que se añada la regla en el fichero file.rules está en 4 SYN/seg, el aspecto del fichero file.rules tras esta prueba es el siguiente:

192.168.1.26 192.168.1.1 2001 80 TCP DROP

Una vez se encuentra la regla en el fichero y ha expirado la regla anterior que dirigía los paquetes hacia h4, se puede observar cómo se descartan dichos paquetes al haberse establecido en el switch la nueva acción para ese flujo en la tabla del flujos OpenFlow. Se observa este efecto al capturar tráfico de red en la interfaz de h4, ya que se dejan de capturar paquetes. Una vez termina el tiempo de expiración de la regla se vuelven a enviar los mensajes al controlador que volverá a poner la misma entrada en la tabla de flujos que en el inicio pero una vez se vuelva a sobrepasar el umbral se volverá a generar la regla indicando que se descarte el tráfico.

4.2.2 Saturación del enlace

En esta prueba se quiere probar que el programa es capaz de escribir en el fichero la regla para descartar los paquetes de un flujo de tráfico, si este puede sobrecargar el enlace. Se considera está situación cuando la tasa de bit/s del flujo supera un 50% del enlace, en este caso 50Mbit/s.
Para esta prueba se genera tráfico TCP entre los hosts h2 y h5. En este caso se quiere generar suficiente tráfico para alcanzar una tasa de 50Mbit/s, para ello se usará iperf [17] del siguiente modo:

- El host h5 estará escuchando tráfico TCP en el puerto 5001 con la opción *iperf -s*
- El host h2 envía tráfico a ese puerto con la opción *iperf –c <IP_dst>*

Como resultado de esta prueba se espera que, si la tasa de bits/s desde h2 a h5 supera el umbral de 50 Mbits/s establecido en el programa, se genere la regla con el formato correcto y que a continuación el switch OpenFlow s2 comience a descartar los paquetes pertenecientes al flujo de tráfico que generó la regla.

El resultado real de esta prueba es el siguiente:

Display

Display filter: ip.dst == 192.168.1.2

Ignored packets: 0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	19962	10051	50,351%	0	0,000%
Between first and last packet	7,977 sec	7,976 sec			
Avg. packets/sec	2502,584	1260,181			
Avg. packet size	4757,566 bytes	9383,280 bytes			
Bytes	94970536	94311350	99,306%	0	0.000%
Avg. bytes/sec	11906209,829	11824632,175			
Avg. MBit/sec	95,250	94,597			

Figura 4-3: Tráfico saturación del enlace

En la Figura 4-3 se puede observar que se han generado 94.597 Mbit/s hacia la IP destino 192.168.1.2 por lo que se debe generar la regla para que se descarten los paquetes de ese flujo. En el fichero file.rules encontramos la siguiente entrada:

192.168.1.21 192.168.1.2 48665 5001 TCP DROP

Una vez se encuentra la regla en el fichero y ha expirado la regla anterior que dirigía los paquetes hacia h5, se puede observar cómo se descartan dichos paquetes al haberse establecido en el switch la nueva acción para ese flujo en la tabla del flujos OpenFlow. Se observa este efecto al capturar tráfico de red en la interfaz de h5, ya que se dejan de capturar paquetes. Una vez termina el tiempo de expiración de la regla se vuelven a enviar los mensajes al controlador que volverá a poner la misma entrada en la tabla de flujos que en el

inicio pero una vez se vuelva a superar el umbral se volverá a generar la regla indicando que se descarte el tráfico.

4.2.3 Tráfico con prioridad

En esta prueba se quiere probar que el programa es capaz de escribir una regla en el fichero para dar prioridad a los flujos de tráfico que tengan una tasa media de bit/s determinada. También se quiere comprobar que el programa acepta distintos protocolos de tráfico y que el controlador actúa de manera consecuente a la regla generada.

Esta prueba también se basa, al igual que la anterior, en la tasa de bits/s promedio. En este caso el umbral para establecer la política de tráfico es un poco menor 20 Mbits/s. La prueba se realizará entre los hosts h3 y h6 y para generar el tráfico necesario, en esta prueba se generará tráfico UDP a diferencia de las dos pruebas anteriores. Para generar el tráfico se usará iperf de la siguiente manera

- El host h6 estará escuchando tráfico TCP en el puerto 5001 con *iperf –s –u –i 1*
- El host h3 envía tráfico a ese puerto con *iperf –c <IP_dst> -u -b <BW>*

Como resultado para esta prueba esperamos que, si la tasa de bit/seg desde h3 a h6 supera el umbral establecido en el programa para la generación de esta regla, se genere una regla con el formato correcto en el fichero file.rules. Dicha regla debe indicar que la acción a realizar por los switch es poner una prioridad baja (TOS 8) y que una vez esté la regla en el fichero, en h6 se reciban los paquetes con dicha prioridad modificada.

El resultado real de esta prueba es el siguiente:

Display

Display filter: ip.dst == 192.168.1.3

Ignored packets: 0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	21222	21222	100,000%	0	0,000%
Between first and last packet	8,256 sec	8,256 sec			
Avg. packets/sec	2570,400	2570,400			
Avg. packet size	1512,000 bytes	1512,000 bytes			
Bytes	32087664	32087664	100,000%	0	0.000%
Avg. bytes/sec	3886444,092	3886444,092			
Avg. MBit/sec	31,092	31,092			

Figura 4-4: Tráfico con prioridad

La regla se debería escribir en el fichero si la tasa de bits está entre 20 Mbits/s y 50 Mbits/s, dado que se ha conseguido generar 31.092 Mbit/s como muestra la Figura 4-4, la regla generada es la siguiente:

192.168.1.24 192.168.1.3 44927 5001 UDP TOS 8

Una vez esté escrita esta regla en el fichero file.rules y haya expirado la entrada actual en la tabla de flujos del switch para ese flujo, que dirigía los paquetes hacia h6, se comienzan a enviar paquetes con TOS 8. En esta prueba a diferencia de las dos anteriores como no se descarta el tráfico, la tasa de bit/s no baja a 0 bit/s y cuando expira la entrada para ese flujo en el controlador como el fichero file.rules se actualiza generando la misma regla y el controlador aplicará la entrada de acuerdo al fichero. Hasta que no termina la comunicación no se vuelven a observar paquetes en el receptor sin TOS 8.

Y en la Figura 4-5 se observa que el paquete desde h3 a h6 se recibe con TOS 8

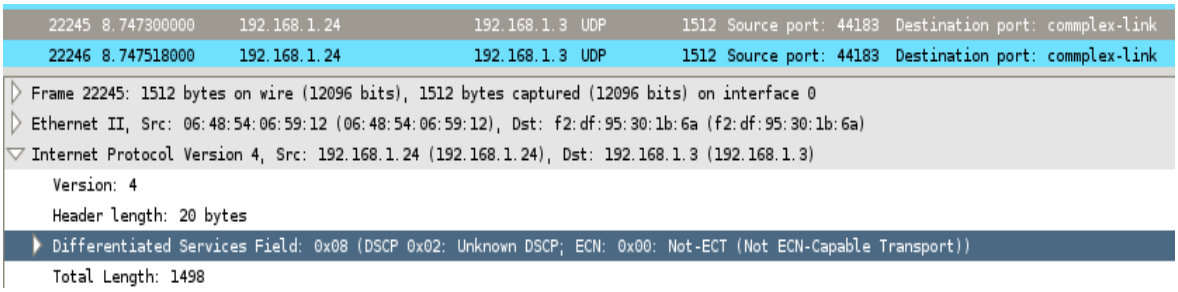


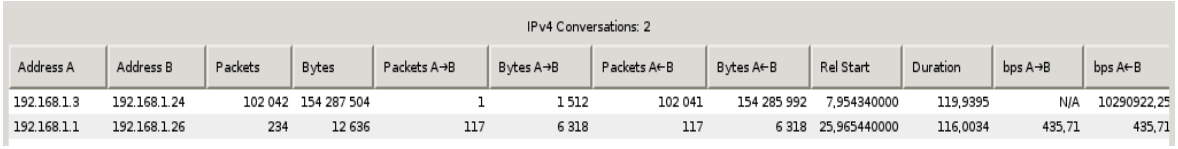
Figura 4-5: Cambia valor TOS de paquetes

4.2.4 Tráfico sin incidencias

En esta prueba se quiere comprobar que no se aplica ninguna política al tráfico cuando no procede, es decir, que si el tráfico en la red no alcanza los umbrales establecidos en el programa, no se generará ninguna regla.

Para la realizar esta prueba se genera tráfico de la misma manera que las los casos 4.1.1 y 4.1.3 pero a tasas menores, para que no se alcance ni umbral de 4 SYN/s y ni el umbral mínimo para la tasa de bit/s de 20 Mbit/s.

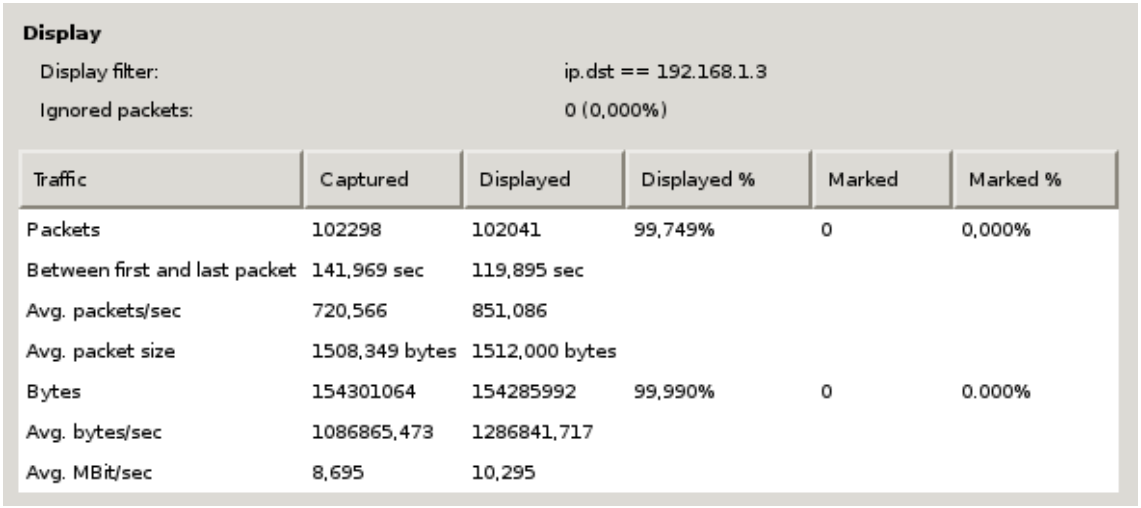
En la Figura 4-6 se observa el tráfico generado para esa prueba en el que encontramos dos conversaciones entre h3 y h6 y entre h1 y h4.


IPv4 Conversations: 2

Address A	Address B	Packets	Bytes	Packets A→B	Bytes A→B	Packets A←B	Bytes A←B	Rel Start	Duration	bps A→B	bps A←B
192.168.1.3	192.168.1.24	102 042	154 287 504	1	1 512	102 041	154 285 992	7,954340000	119,9395	N/A	10290922,25
192.168.1.1	192.168.1.26	234	12 636	117	6 318	117	6 318	25,965440000	116,0034	435,71	435,71

Figura 4-6: Trafico sin incidencias

Filtrando por la IP destino de h6, vemos que la tasa media en es aproximadamente 10 Mbit/s como muestra la Figura 4-7, lo que no se considera ni una posible saturación del enlace ni prioriza el tráfico.



Display

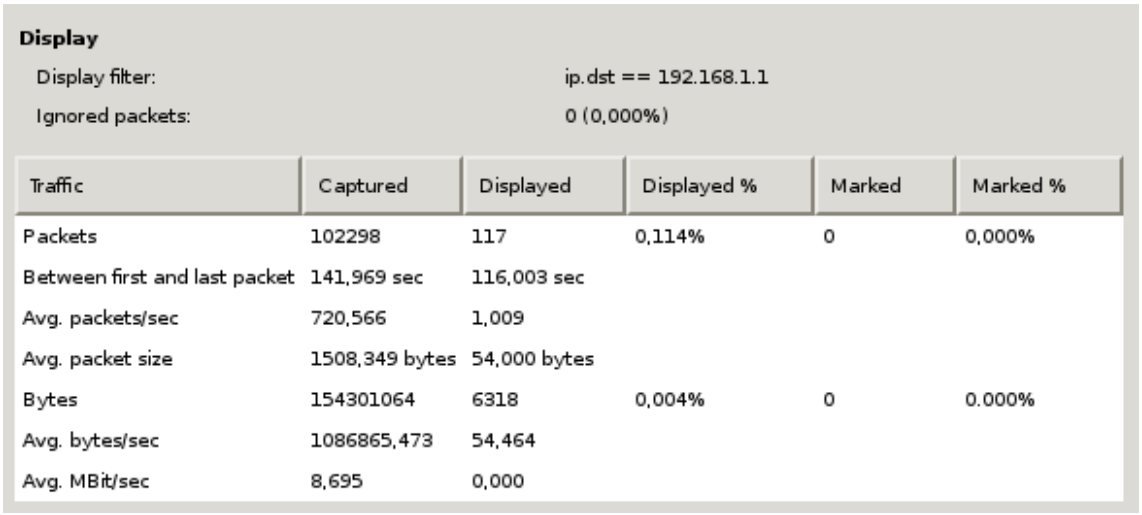
Display filter: ip.dst == 192.168.1.3

Ignored packets: 0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	102298	102041	99,749%	0	0,000%
Between first and last packet	141,969 sec	119,895 sec			
Avg. packets/sec	720,566	851,086			
Avg. packet size	1508,349 bytes	1512,000 bytes			
Bytes	154301064	154285992	99,990%	0	0.000%
Avg. bytes/sec	1086865,473	1286841,717			
Avg. MBit/sec	8,695	10,295			

Figura 4-7: Tráfico sin incidencia Mbit/s

Filtrando por IP destino de h4 vemos que en la comunicación de media se ha enviado 1 paquete/s como muestra la Figura 4-8 por lo que no se considera un ataque por inundación de SYNs



Display

Display filter: ip.dst == 192.168.1.1

Ignored packets: 0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	102298	117	0,114%	0	0,000%
Between first and last packet	141,969 sec	116,003 sec			
Avg. packets/sec	720,566	1,009			
Avg. packet size	1508,349 bytes	54,000 bytes			
Bytes	154301064	6318	0,004%	0	0.000%
Avg. bytes/sec	1086865,473	54,464			
Avg. MBit/sec	8,695	0,000			

Figura 4-8 Tráfico sin incidencias SYN

Como se esperaba, ya que en la Figura 4-7 y Figura 4-8 no se alcanza ninguno de los umbrales, en esta prueba no se ha escrito ninguna regla en el fichero file.rules y por lo tanto no se ha descartado ningun paquete ni se ha modificado la prioridad.

4.2.5 Acciones PUSH_VLAN, OUTPUT y FLOOD

El analizador de tráfico solo genera reglas indicando la aplicación de dos de las cinco acciones que puede realizar el controlador implementado: DROP y SET_TOS. En esta prueba se va a generar manualmente el fichero file.rules con las tres acciones restantes y se va a generar el tráfico que se indique en el fichero, para comprobar que se realizan las acciones correctamente.

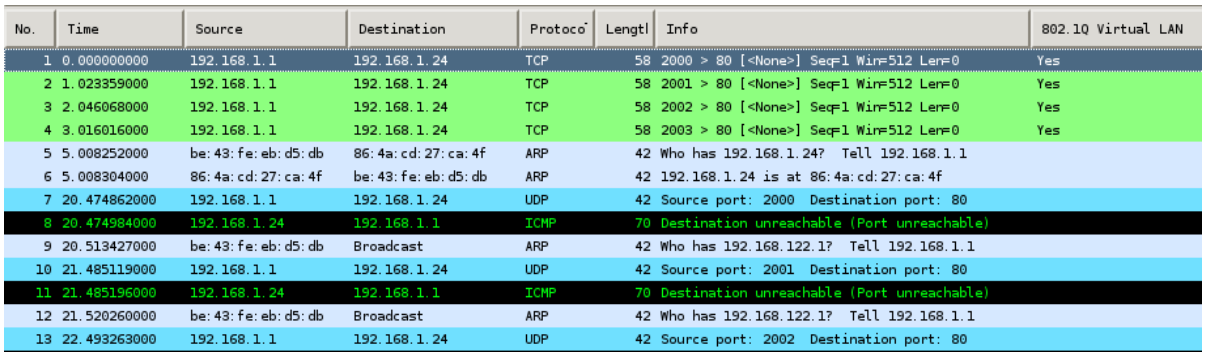
Para comprobar la acción PUSH_VLAN se define la siguiente regla:

*192.168.1.1 192.168.1.24 * * TCP PUSH_VLAN 20*

Se aplicará la acción a todo el tráfico TCP desde h4 a h3, los puertos origen y destino de los paquetes podrán tener cualquier valor. Para realizar esta prueba se genera tráfico TCP y UDP desde h3 con hping3, para ver la diferencia ya que solo al tráfico TCP se le debe aplicar la acción:

- *hping3 <IP_dst> -s <Port_src> -p <Port_dst>*
- *hping3 <IP_dst> --udp -s <Port_src> -p <Port_dst>*

Para ver los resultados de esta prueba capturamos el tráfico que recibe h3



No.	Time	Source	Destination	Protocol	Length	Info	802.1Q Virtual LAN
1	0.000000000	192.168.1.1	192.168.1.24	TCP	58	2000 > 80 [<None>] Seq=1 Win=512 Len=0	Yes
2	1.023359000	192.168.1.1	192.168.1.24	TCP	58	2001 > 80 [<None>] Seq=1 Win=512 Len=0	Yes
3	2.046068000	192.168.1.1	192.168.1.24	TCP	58	2002 > 80 [<None>] Seq=1 Win=512 Len=0	Yes
4	3.016016000	192.168.1.1	192.168.1.24	TCP	58	2003 > 80 [<None>] Seq=1 Win=512 Len=0	Yes
5	5.008252000	be:43:fe:eb:d5:db	86:4a:cd:27:ca:4f	ARP	42	Who has 192.168.1.24? Tell 192.168.1.1	
6	5.008304000	86:4a:cd:27:ca:4f	be:43:fe:eb:d5:db	ARP	42	192.168.1.24 is at 86:4a:cd:27:ca:4f	
7	20.474862000	192.168.1.1	192.168.1.24	UDP	42	Source port: 2000 Destination port: 80	
8	20.474984000	192.168.1.24	192.168.1.1	ICMP	70	Destination unreachable (Port unreachable)	
9	20.513427000	be:43:fe:eb:d5:db	Broadcast	ARP	42	Who has 192.168.122.1? Tell 192.168.1.1	
10	21.485119000	192.168.1.1	192.168.1.24	UDP	42	Source port: 2001 Destination port: 80	
11	21.485196000	192.168.1.24	192.168.1.1	ICMP	70	Destination unreachable (Port unreachable)	
12	21.520260000	be:43:fe:eb:d5:db	Broadcast	ARP	42	Who has 192.168.122.1? Tell 192.168.1.1	
13	22.493263000	192.168.1.1	192.168.1.24	UDP	42	Source port: 2002 Destination port: 80	

Figura 4-9: Acción PUSH_VLAN

Como vemos en la Figura 4-9, solo el tráfico TCP (en verde) tiene cabecera VLAN. Además si nos fijamos en la imagen, la acción se aplica a los 4 paquetes TCP aunque los puertos origen son distintos para cada paquete (2000, 2001, 2002 y 2003).

Para comprobar la acción OUTPUT se define la siguiente regla:

*192.168.1.26 192.168.1.2 * * TCP OUTPUT 4*

Se aplicará la acción a todo el tráfico TCP desde h1 a h5, esta acción hará que los paquetes destinados a h5 no lleguen a dicho destino y se desvíen a h4. Para realizar esta prueba se genera tráfico TCP desde h1 a h5 con hping3.

Para comprobar el resultado se hará uso de la herramienta tcpdump iniciada en h4 y h5

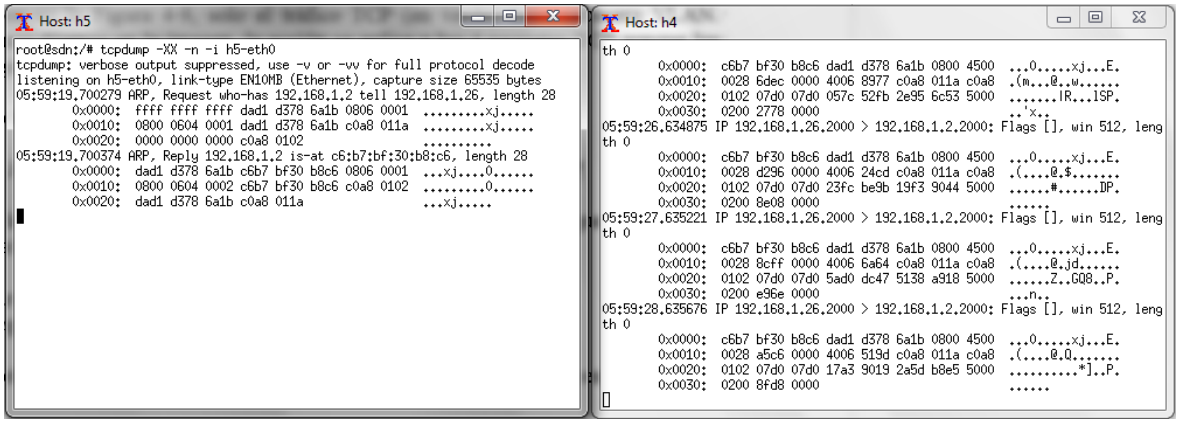


Figura 4-10: Acción OUTPUT

En la Figura 4-10 se puede observar como h4 captura paquetes IP con destino 192.168.1.2 (h5), mientras que h5 solo ha capturado los paquetes ARP.

Para comprobar la acción FLOOD se define la siguiente regla:

*192.168.1.1 192.168.1.24 * 25 TCP FLOOD*

Se aplicará la acción al tráfico desde h4 a h3, desde cualquier puerto origen y con puerto destino 25. Para realizar esta prueba se genera tráfico TCP desde h4 a h3 con hping3.

Para comprobar el resultado en este caso se analizaran los mensajes que se envían el controlador y el switch a través del protocolo OpenFlow para establecer una nueva entrada en la tabla de flujos.

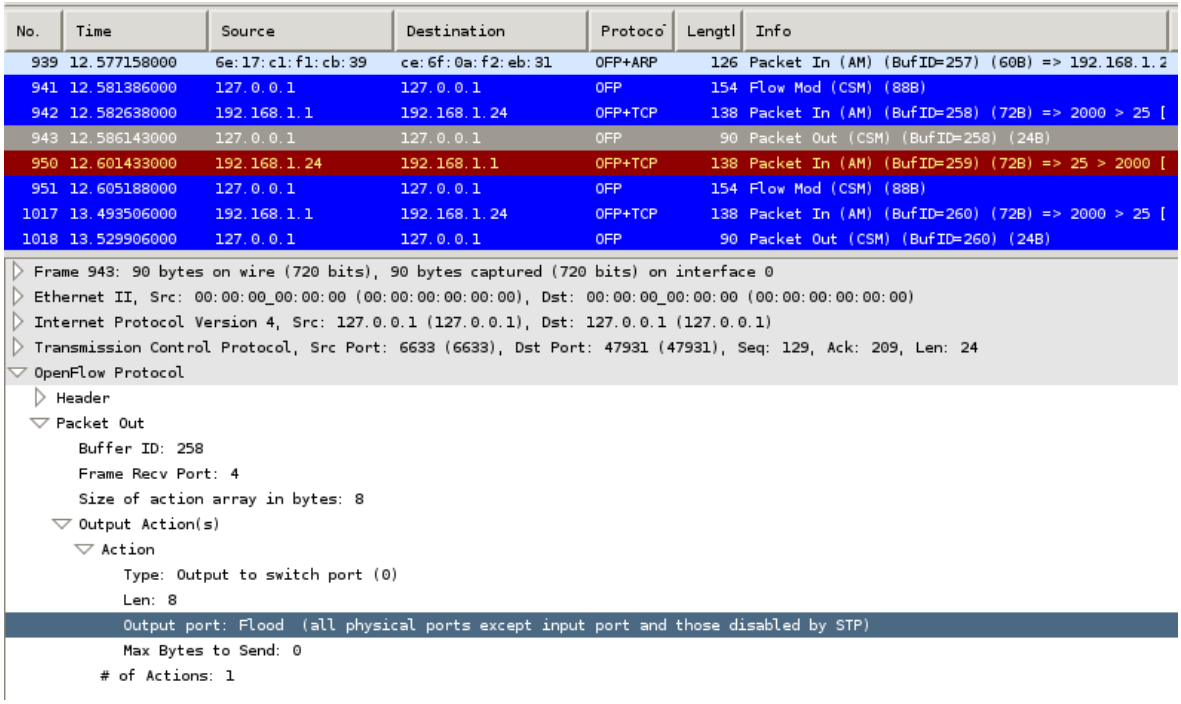


Figura 4-11: Acción FLOOD

El paquete número 942 de la Figura 4-11, es un paquete de tipo Packet In que envía el switch al controlador cuando recibe un paquete que no tiene ninguna coincidencia en su tabla de flujos, como vemos se trata del paquete desde h4 a h3 y puerto destino 25. A este mensaje el controlador responde con un mensaje Packet Out, que es el que está seleccionado en la captura de Wireshark y vemos que como parte del protocolo OpenFlow se especifica que la acción es enviar el paquete por “el puerto Flood”.

4.3 Pruebas de rendimiento

Con estas pruebas se quiere comprobar el número de bit/s y paquetes/s que es capaz de procesar el analizador de tráfico en su funcionamiento normal. Para ello se utilizan los datos que recoge el programa de número de bits de los paquetes y número de paquetes para calcular dichos valores en intervalos de 5 segundos. Para la prueba se ha hecho una ejecución de 15 minutos y se ha obtenido la media y desviación estándar de estos valores que se muestra en la Figura 4-12 y en la Figura 4-13.

Para la tasa real de procesamiento del analizador de tráfico no quede enmascarada por el ancho de banda de los enlaces, vamos a aumentar el ancho de banda de los enlaces hasta el máximo permito por Mininet, un valor de 10Gbit/s.

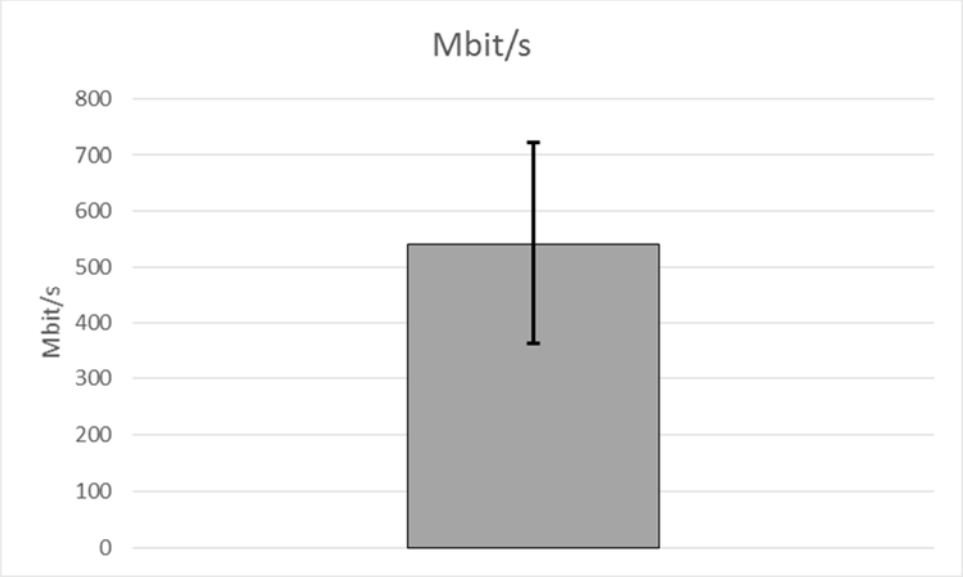


Figura 4-12: Tasa de procesamiento en Mbit/s

Figura 4-13: Tasa de procesamiento en paquetes/s

Como se puede observar se ha conseguido una tasa de procesamiento media de 542 Mbit/s con una variación del 33% y 36250 paquetes/s con una variación del 11%. Teniendo en cuenta que la capacidad máxima del enlace es de 10 Gbit/s; la tasa de procesamiento obtenida es un dato bastante razonable, puesto que nos encontramos en un entorno doblemente virtualizado, ya que Mininet genera una red virtual y a su vez Mininet se está ejecutando en una máquina virtual en nuestro caso. Además de la doble virtualización, la interacción entre el controlador POX y el analizador de tráfico a través de un fichero de texto ralentiza la tasa de procesamiento de este último.

5 Conclusiones y trabajo futuro

Dada la evolución que están tomando las aplicaciones y tecnologías que se sustentan sobre redes de computadoras, las redes definidas por software resultan una buena solución para dar cabida a todas las nuevas tendencias que están surgiendo en los últimos años.

Las SDN con la separación del plano de control y de datos facilitan el mantenimiento y gestión de las redes y permiten un fácil desarrollo de aplicaciones de red, como pueden ser sistemas de monitorización o seguridad.

En este trabajo se ha desarrollado un sistema de monitorización para SDN. El sistema diseñado permite monitorizar todo el tráfico de la red para obtener información de dicho tráfico para analizarlo y aplicar políticas en función de las necesidades de seguridad y calidad de servicio de la red y así asegurarnos de su correcto funcionamiento.

Nuestro sistema utiliza un elemento independiente al controlador para analizar el tráfico. De esta manera aunque la tasa de procesamiento se pueda ver ralentizada debido a una comunicación poco eficiente entre el analizador de tráfico y el controlador, nos aseguramos de no sobrecargar el controlador. El analizador de tráfico independiente también permite realizar estimaciones más precisas sobre el tráfico que circula por la red ya que analiza todo el tráfico y no una muestra del mismo o solo los flujos activos en el switch.

El sistema desarrollado, funciona de manera adecuada en una topología de red de reducido tamaño emulada en Mininet. Se han obtenido los resultados esperados en una serie de situaciones para las que el sistema está diseñado. Consideramos que este sistema es bastante eficiente, consiguiendo una tasa de procesamiento media de 542 Mbit/s y 36,250 paquetes/s.

Puesto que las SDN basadas en OpenFlow y concretamente los controladores basados en POX, se pueden desplegar en muchas redes existentes, tanto físicas como virtuales la implantación de este sistema sería directa., Y, utilizando Mininet que provee virtualización ligera de cada elemento de red, hemos podido crear topologías realistas de manear interactiva, usando el mismo código que se usaría en una red real. Nuestro sistema se podría emplear en una red real sin ninguna modificación.

Como conclusión personal, este trabajo me ha permitido profundizar en el conocimiento de las SDN y en particular comprender la importancia de la monitorización para asegurar el correcto funcionamiento de las redes actuales. Con este trabajo he podido comprobar que con esta nueva arquitectura no hace falta implementar sistemas y aplicaciones excesivamente complejos para gestionar la red. La arquitectura SDN junto con el protocolo OpenFlow y el controlador POX me han permitido desarrollar un sistema de monitorización desarrollado en Python y lenguaje C, y emularlo en un entorno realista (Mininet), y me ha dado la posibilidad de usar herramientas de red como iperf o hping3.

5.1 Trabajo futuro

Durante el desarrollo del sistema se han identificado una serie de posibilidades que en su momento se decidió, por motivos de tiempo y complejidad, no incorporar al mismo, y así dejarlas para una incorporación en una versión futura.

El sistema actual solo tiene tres tipos de situaciones ante las que actúa. En un futuro se puede plantear, incorporar la detección de otro tipo de ataques o incidencias comunes en la red, proporcionando así más seguridad y calidad de servicio a las redes en las que se implante dicho sistema.

Uno de dichos ataques puede ser un ataque distribuido de denegación de servicio (DDoS), el programa actual solo genera reglas desde una IP y puerto origen concreto a una IP y puerto destino, aunque el controlador si se ha desarrollado de manera que, por ejemplo, pueda aplicar reglas hacia un destino desde varios orígenes si en el fichero file.rules como IP origen encuentra el carácter "*". Existen trabajos relacionados que presentan esquemas para la detección de ataques DDoS desarrollados como una aplicación de controlador SDN [18] que bloquean ataques que los sistemas tradicionales de detección de anomalías no soportan pero aun presentan carencias en cuanto a funcionalidad

Otra mejora que se puede desarrollar es la generación de reglas correctas para flujos de tráfico con paquetes fragmentados, ya que puesto que la estructura de estos paquetes difiere de la de un paquete no fragmentado el sistema actual genera la regla de manera errónea.

En cuanto al trabajo futuro en la evaluación de prestaciones se plantea la evaluación del número de flujos o reglas OpenFlow que puede soportar el sistema. En principio este parámetro es dependiente del equipamiento de red usado, pero el primer paso debe ser evaluar el rendimiento del switch OpenFlow que viene por defecto en Mininet.

Por último se plantea combinar los dos elementos del sistema ya que la comunicación entre ambos mediante el fichero file.rules es poco eficiente. Esto aceleraría la detección de las distintitas situaciones y la aplicación de las políticas al tráfico. Para ello se propone el uso del controlador NOX que permite desarrollar programas en C++ o la implementación de un sistema ligero de mensajes OpenFlow que nos permita reducir al máximo la comunicación y sobrecarga de procesamiento producida por los frameworks de los controladores. De acuerdo a los desarrolladores de NOX y POX [19] el rendimiento de NOX es mayor soportando 50,000 flujos por segundo mientras que POX soporta 30,000.

Referencias

[1] KIM, Hyojoon; FEAMSTER, Nick. Improving network management with software defined networking. *Communications Magazine, IEEE*, 2013, vol. 51, no 2, p. 114-119.

[2] BOZAKOV, Zdravko; PAPADIMITRIOU, Panagiotis. Towards a scalable software-defined network virtualization platform. En *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014. p. 1-8.

[3] FEAMSTER, Nick; REXFORD, Jennifer; ZEGURA, Ellen. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 2014, vol. 44, no 2, p. 87-98.

[4] FUNDATION, Open Networking. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[5] JARSCHEL, Michael, et al. Interfaces, attributes, and use cases: A compass for SDN. *Communications Magazine, IEEE*, 2014, vol. 52, no 6, p. 210-217.

[6] MCKEOWN, Nick, et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008, vol. 38, no 2, p. 69-74.JARSCHEL, Michael, et al. Interfaces, attributes, and use cases: A compass for SDN. *Communications Magazine, IEEE*, 2014, vol. 52, no 6, p. 210-217.

[7] VAUGHAN-NICHOLS, Steven J. OpenFlow: The next generation of the network?. *Computer*, 2011, no 8, p. 13-15.

[8] VAN DER POL, Ronald. D1. 2 OpenFlow. 2011.

[9] KHONDOKER, Rahamatullah, et al. Feature-based comparison and selection of Software Defined Networking (SDN) controllers. En *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*. IEEE, 2014. p. 1-7.

[10] SUH, Michelle, et al. Building firewall over the software-defined network controller. En *Advanced Communication Technology (ICACT), 2014 16th International Conference on*. IEEE, 2014. p. 744-748.

[11] CHOWDHURY, Shubhajit Roy, et al. Payless: A low cost network monitoring framework for software defined networks. En *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014. p. 1-9.

[12] BRAGA, Rodrigo; MOTA, Edjard; PASSITO, Alexandre. Lightweight DDoS flooding attack detection using NOX/OpenFlow. En *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*. IEEE, 2010. p. 408-415.

[13] SHIN, Seungwon; GU, Guofei. CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). En Network Protocols (ICNP), 2012 20th IEEE International Conference on. IEEE, 2012. p. 1-6.

[14] SHIN, Seungwon, et al. FRESCO: Modular Composable Security Services for Software-Defined Networks. En NDSS. 2013.

[15] LANTZ, Bob; HELLER, Brandon; MCKEOWN, Nick. A network in a laptop: rapid prototyping for software-defined networks. En *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010. p. 19.

[16] http://linux.die.net/man/8/hping3 5 de Mayo de 2015

[17] https://iperf.fr/ 5 de Mayo de 2015

[18] LIM, Sharon, et al. A SDN-oriented DDoS blocking scheme for botnet-based attacks. En *Ubiquitous and Future Networks (ICUFN), 2014 Sixth International Conf on*. IEEE, 2014. p. 63-68.

[19] http://www.noxrepo.org/pox/about-pox/ 1 de Junio de 2015

Glosario

SDN: Red definida por Software (Software Defined Network), arquitectura de red que se basa en la separación de los planos de control y datos.

QoS: Calidad de Servicio (Quality of Service).

DoS: Denegación de Servicio (Denial of Service), un ataque DoS es un ataque a un sistema de computadoras o red que causa que un servicio o recurso sea inaccesible a los usuarios legítimos.

DDoS: Denegación de Servicio Distribuido (Distributed Denial of Service), una variante del ataque DoS, se realiza de manera distribuida desde varios puntos de conexión.

TOS: Tipo de Servicio (Type of Service), es un campo de la cabecera IP (versión 4), que permite determinar la prioridad de un paquete.

SYN: bit de sincronización (**syn**chronization), es un bit de control dentro del segmento TCP, que se utiliza para sincronizar los números de secuencia iniciales ISN de una conexión en el procedimiento de establecimiento de tres fases

UDP: Protocolo de Datagramas de Usuario (User Datagram Protocol), es un protocolo de la capa de transporte que no garantiza la entrega de los paquetes.

TCP: Protocolo de Control de Transmisión (Transmission Control Protocol), es un protocolo de la capa de transporte usado por las aplicacioens que requieren una entrega garantizada.

VLAN: Red de Area Local Virtual (Virtual Local Area Network), agrupación de host en una o varias LAN que permite la comunicación entre host como si estuvieran en la misma LAN física.

API: Interfaz de Programación de Aplicaciones (Application Programming Interface), es un conjunto de funciones o métodos usados para acceder a ciertas funcionalidades.

ISP: Proveedores de Servicios de Internet (Internet Service Provider)

IaaS: Infraestructura como Servicio (Infrastructure as a Service), hace referencia a los servicios cloud que proporcionan acceso a recursos informáticos situados en la nube.

IDS: Sistema de Detección de Intrusiones (Intrusion Detection System) es un programa de detección de accesos no autorizados en un ordenador o red

Anexo I: Código del controlador

```
from pox.core import core
from pox.lib.packet.ipv4 import ipv4
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str
from pox.lib.util import str_to_bool
from pox.lib.addresses import IPAddr
import time
import threading

campo={}
protocolo={}
numLines = 0

def leeReglas():
    global campo
    global protocolo
    global numLines
    var = 1

    while (var == 1):
        #abrir fichero file.rules
        infile = open('file.rules','r')

        i=0
        for line in infile:

            campo[i] = (line).strip().split("\t") #campo es variable global

            if campo[i][4] == "TCP":
                protocolo[i] = ipv4.TCP_PROTOCOL
            elif campo[i][4] == "UDP":
                protocolo[i] = ipv4.UDP_PROTOCOL
            elif campo[i][4] == "ICMP":
                protocolo[i] = ipv4.ICMP_PROTOCOL
            elif campo[i][4] == "IGMP":
                protocolo[i] = ipv4.IGMP_PROTOCOL
            else:
                pass

            i+=1

    infile.close()
    numLines=i

    #print("Ejecutando hilo")
    time.sleep(5)
```

```
    return


class LearningSwitch (object):

  def __init__ (self, connection, transparent):
    # Switch we'll be adding L2 learning switch capabilities to
    self.connection = connection
    self.transparent = transparent

    global campo
    global protocolo
    global numLines


    # Our table
    self.macToPort = {}

    # We want to hear PacketIn messages, so we listen
    # to the connection
    connection.addListeners(self)

    # We just use this to know when to log a helpful message
    self.hold_down_expired = _flood_delay == 0

    log.debug("Initializing LearningSwitch, transparent=%s",
              str(self.transparent))

  def _handle_PacketIn (self, event):
    """
    Handle packet in messages from the switch to implement above algorithm.
    """

    packet = event.parsed

    def flood (message = None):
      """ Floods the packet """
      msg = of.ofp_packet_out()
      if time.time() - self.connection.connect_time >= _flood_delay:
        # Only flood if we've been connected for a little while...

        if self.hold_down_expired is False:
          # Oh yes it is!
          self.hold_down_expired = True
          log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

        if message is not None: log.debug(message)
        log.debug("%i: flood %s -> %s", event.dpid,packet.src,packet.dst)
        # OFPP_FLOOD is optional; on some switches you may need to change
```

```
    # this to OFPP_ALL.
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
  else:
    pass
    log.info("Holding down flood for %s", dpid_to_str(event.dpid))
  msg.data = event.ofp
  msg.in_port = event.port
  self.connection.send(msg)

def drop (duration = None):
  """
  Drops this packet and optionally installs a flow to continue
  dropping similar ones for a while
  """

  if duration is not None:
    if not isinstance(duration, tuple):
      duration = (duration,duration)
    msg = of.ofp_flow_mod()
    msg.match = of.ofp_match.from_packet(packet)
    msg.idle_timeout = duration[0]
    msg.hard_timeout = duration[1]
    msg.buffer_id = event.ofp.buffer_id
    self.connection.send(msg)
  elif event.ofp.buffer_id is not None:
    msg = of.ofp_packet_out()
    msg.buffer_id = event.ofp.buffer_id
    msg.in_port = event.port
    self.connection.send(msg)

def push_vlan (vlan_id,port):
                vlan_action = of.ofp_action_vlan_vid()
                vlan_action.vlan_vid = vlan_id

                log.debug("push_vlan")
                msg = of.ofp_flow_mod()
                msg.match = of.ofp_match.from_packet(packet, event.port)
                msg.idle_timeout = 10
                msg.hard_timeout = 30
                msg.actions.append(vlan_action)
                msg.actions.append(of.ofp_action_output(port = port))
                msg.actions.append(of.ofp_action_output(port = 1))
                msg.data = event.ofp # 6a
                self.connection.send(msg)

def set_tos (tos,port):
                nw_action = of.ofp_action_nw_tos()
                nw_action.nw_tos = tos

                log.debug("set_tos")
```

```
                msg = of.ofp_flow_mod()
                msg.match = of.ofp_match.from_packet(packet, event.port)
                msg.idle_timeout = 10
                msg.hard_timeout = 30
                msg.actions.append(nw_action)
                msg.actions.append(of.ofp_action_output(port = port))
    msg.actions.append(of.ofp_action_output(port = 1))
                msg.data = event.ofp
                self.connection.send(msg)

def output (port):
                log.debug("output")
                msg = of.ofp_flow_mod()
                msg.match = of.ofp_match.from_packet(packet, event.port)
                msg.idle_timeout = 10
                msg.hard_timeout = 30
                msg.actions.append(of.ofp_action_output(port = port))
    msg.actions.append(of.ofp_action_output(port = 1))
                msg.data = event.ofp
                self.connection.send(msg)


self.macToPort[packet.src] = event.port # 1

if not self.transparent: # 2
  if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
    drop() # 2a
    return

if packet.dst.is_multicast:
  flood() # 3a
else:
  if packet.dst not in self.macToPort: # 4
    flood("Port for %s unknown -- flooding" % (packet.dst,)) # 4a
  else:
    port = self.macToPort[packet.dst]
    if port == event.port: # 5
      # 5a
      log.warning("Same port for packet from %s -> %s on %s.%s.  Drop."
          % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
      drop(10)
      return
```

```
#comprobar condiciones fichero file.rules

for i in range(numLines):

    if packet.type == packet.IP_TYPE: #IP_TYPE
        ip_packet = packet.payload
        tcp_packet = ip_packet.payload

        if campo[i][0] == "*" or ip_packet.srcip == IPAddr(campo[i][0]):

            if campo[i][1] == "*" or ip_packet.dstip == IPAddr(campo[i][1]):

                if campo[i][4] == "*" or ip_packet.protocol == protocolo[i]:

                    if campo[i][3] == "*" or tcp_packet.dstport == int(campo[i][3]):

                        if campo[i][2] == "*" or tcp_packet.srcport == int(campo[i][2]):

                            if campo[i][5] == "FLOOD":
                                log.debug("flood")
                                flood()
                                return

                            elif campo[i][5] == "DROP":
                                log.debug("drop")
                                drop(30)
                                return

                            elif campo[i][5] == "SET_TOS":
                                log.debug("set tos")
                                set_tos(int(campo[i][6]),port)
                                return
                            elif campo[i][5] == "PUSH_VLAN":
                                log.debug("push_vlan")
                                push_vlan(int(campo[i][6]),port)
                                return
                            elif campo[i][5] == "OUTPUT":
                                log.debug("campo output")
                                output(int(campo[i][6]))
                                return

#si no hay regla se envia normalmente
log.debug("installing flow for %s.%i -> %s.%i" %
   (packet.src, event.port, packet.dst, port))
msg = of.ofp_flow_mod()
msg.match = of.ofp_match.from_packet(packet, event.port)
msg.idle_timeout = 2 #10
msg.hard_timeout = 2 #30
msg.actions.append(of.ofp_action_output(port = port))
```

```
        msg.actions.append(of.ofp_action_output(port = 1))
           msg.data = event.ofp # 6a
        self.connection.send(msg)

class l2_learning (object):
  """
  Waits for OpenFlow switches to connect and makes them learning switches.
  """
  def __init__ (self, transparent):
    core.openflow.addListeners(self)
    self.transparent = transparent

  def _handle_ConnectionUp (self, event):
    h = threading.Thread(target=leeReglas)
    h.start()
    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)


def launch (transparent=False, hold_down=_flood_delay):
  """
  Starts an L2 learning switch.
  """
  try:
    global _flood_delay
    _flood_delay = int(str(hold_down), 10)
    assert _flood_delay >= 0
  except:
    raise RuntimeError("Expected hold-down to be a number")

  core.registerNew(l2_learning, str_to_bool(transparent))
```

Anexo II: Código del analizador de tráfico

```
#include <stdio.h>
#include <pcap.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <stdint.h>

struct Trama
{
   uint8_t IPsrc[4];
   uint8_t IPdst[4];
   uint8_t PortDst[2];
   uint8_t PortSrc[2];
   uint8_t proto[1];
   struct timeval timeStamp;
   uint8_t ToS[1];
   int8_t syn;//bit de control del segmento tcp,se utiliza para sincronizar losnumeros de
secuencia iniciales de una conexion en el procedimiento de tres fases
   int synseg;//contador para los syns por seg
   uint32_t bitseg; //contador para los bits por seg
   uint32_t len;// lo guardo en bits el pcap lo da en byte

};

pthread_mutex_t lockEnd = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lockTabla = PTHREAD_MUTEX_INITIALIZER;

int addTrama(struct Trama trama);
int vacio(struct Trama  trama);
int compararContenido(struct Trama tramaTabla, struct Trama trama);
void * comprobarBitseg(void* a);

#define TAM_TABLA 1000
#define MAX_SYN 40  // 40 syn/10seg <--> 4 syn/seg
#define MAX_BITSEG_DROP 500000000 //500 000000b/10seg <--> 50 Mb/seg ---50%
de 100Mb/s
#define MAX_BITSEG_TOS  200000000//200 000000/10 seg <-->20 Mb/seg ---20% de
10Mb/s

int tam_tabla = TAM_TABLA;
struct Trama *tabla=NULL;
uint8_t end;
```

```
void * comprobarBitseg(void* a)
{
   int i;
   int j=0;
   FILE *fp;
   char protocolo[3];
   memset(protocolo,'0',3);


   printf("hilo iniciado\n");
   while(1)
   {

   pthread_mutex_lock(&lockEnd);
   if(end==0)
   {
         //fclose(fp);
      break;
   }
   pthread_mutex_unlock(&lockEnd);

   fp = fopen("file.rules","w");
   printf("fichero file.rules\n");


   for(i=0; i<tam_tabla; i++)
      {

      pthread_mutex_lock(&lockTabla);

         //Nuúmero de paquetes TCP SYN por segundo que recibe una dirección IP destino
desde una direccion IP origen.-->DROP
         if(tabla[i].synseg > MAX_SYN)
         {
            if(*tabla[i].proto == 6){
               memcpy(protocolo,"TCP",3);
            }else if(*tabla[i].proto == 17){
               memcpy(protocolo,"UDP",3);
            }else{
               memcpy(protocolo,"*",1);
            }

            fprintf(fp,"%d.%d.%d.%d\t%d.%d.%d.%d\t%d\t%d\t%s\tDROP\n"
                 , tabla[i].IPsrc[0],tabla[i].IPsrc[1],tabla[i].IPsrc[2],tabla[i].IPsrc[3]
                 , tabla[i].IPdst[0],tabla[i].IPdst[1],tabla[i].IPdst[2],tabla[i].IPdst[3]
                 ,(tabla[i].PortSrc[0]<<8)+ tabla[i].PortSrc[1]
                 ,(tabla[i].PortDst[0]<<8)+ tabla[i].PortDst[1]
                 ,protocolo);
            fflush(fp);
```

```
            printf("regla 1\n");

        }
        //Tasa en bits/s de flujo supera el 50% de la capacidad del enlace -->DROP
        else if(tabla[i].bitseg > MAX_BITSEG_DROP)
        {
            if(*tabla[i].proto == 6){
                memcpy(protocolo,"TCP",3);
            }else if(*tabla[i].proto == 17){
                memcpy(protocolo,"UDP",3);
            }else{
                memcpy(protocolo,"*",1);
            }

             fprintf(fp,"%d.%d.%d.%d\t%d.%d.%d.%d\t%d\t%d\t%s\tDROP\n"
                  , tabla[i].IPsrc[0],tabla[i].IPsrc[1],tabla[i].IPsrc[2],tabla[i].IPsrc[3]
                  , tabla[i].IPdst[0],tabla[i].IPdst[1],tabla[i].IPdst[2],tabla[i].IPdst[3]
                  ,(tabla[i].PortSrc[0]<<8)+ tabla[i].PortSrc[1]
                  ,(tabla[i].PortDst[0]<<8)+ tabla[i].PortDst[1]
                  ,protocolo);
            fflush(fp);
            printf("regla 2\n");

        }
        //Tasa de bits/s del flujo superan el 20%  de la capacidad del enlace-->prioridad
baja (TOS 0)
        else if(tabla[i].bitseg > MAX_BITSEG_TOS && tabla[i].bitseg
<MAX_BITSEG_DROP)
        {
            if(*tabla[i].proto == 6){
                memcpy(protocolo,"TCP",3);
            }else if(*tabla[i].proto == 17){
                memcpy(protocolo,"UDP",3);
            }else{
                memcpy(protocolo,"*",1);
            }

        fprintf(fp,"%d.%d.%d.%d\t%d.%d.%d.%d\t%d\t%d\t%s\tTOS\t8\n"
                  , tabla[i].IPsrc[0],tabla[i].IPsrc[1],tabla[i].IPsrc[2],tabla[i].IPsrc[3]
                  , tabla[i].IPdst[0],tabla[i].IPdst[1],tabla[i].IPdst[2],tabla[i].IPdst[3]
                  ,(tabla[i].PortSrc[0]<<8)+ tabla[i].PortSrc[1]
                  ,(tabla[i].PortDst[0]<<8)+ tabla[i].PortDst[1]
                  ,protocolo);
            fflush(fp);
            printf("regla 3\n");

        }
        tabla[i].bitseg = 0;
        tabla[i].synseg = 0;
```

```
            pthread_mutex_unlock(&lockTabla);

        }

        sleep(10);//cambia a 10 seg
        fclose(fp);

    }

}

int main(int argc, char** argv){

   struct pcap_pkthdr header;
   const uint8_t *packet;
   uint8_t MacDestino[6];
   uint8_t MacOrigen[6];
   uint8_t Ethertype[2];
   uint8_t IPsrc[16];
   uint8_t IPver[1];
   uint8_t ofType[1];
   uint8_t  ethType[2];
   uint8_t ToS_aux[2];
   uint8_t flags[1];
   int8_t ToS;
   int8_t ihl;
   int8_t syn_aux;
   struct Trama trama;
   int j=0;
   int i=1;


   pthread_t id;
   int ret;

           uint32_t len2 =0;

           struct timeval tv_ini;
           struct timeval tv_fin;
           int count;


   //abrir fichero pcap
   pcap_t *handle;
   char errbuf[PCAP_ERRBUF_SIZE];
    //parametro de entrada del programa que dispositivo se escucha
   char *dev = argv[1];

    printf("Device: %s\n", dev);
```

```
  //procesar paquetes

  //reserva de memoria para la tabla
  tabla = (struct Trama *) malloc(tam_tabla*sizeof(struct Trama));


   end=1;
   //creo hilo para contador
   ret=pthread_create(&id, NULL, comprobarBitseg,NULL);
   if(ret!=0)
   {
      printf("Error al crear hilo");
      exit(0);
   }


   count =0;
   len2 = 0;
   gettimeofday(&tv_ini, NULL);

  handle = pcap_open_live(dev, BUFSIZ, 0, -1, errbuf);
  //handle = pcap_open_offline("cap3.pcap", errbuf);
  if (handle == NULL)
  {
    fprintf(stderr,"pcap_open: %s\n", errbuf);
   exit(0);

  }
while(1)
{
   if((packet = pcap_next(handle,&header)))
  {
       count++; //contador del numero de paquetes que llegan al programa
       trama.timeStamp = header.ts;
       trama.len = header.len*8;//se pasan bytes a bits
       len2+=(header.len*8);

     //Mac dsetino origen y ethertype
     //GUARDANDO DATOS EN VARIABLES
      memcpy(MacDestino,packet, 6);
      memcpy(MacOrigen,&packet[6], 6);
      memcpy(Ethertype,&packet[12], 2);
      memcpy(IPver, &packet[14], 1);
```

```
      if(((*IPver)>>4) == 0x4)//IP version 4
      {

         memcpy(trama.IPsrc,&packet[26], 4);
         memcpy(trama.IPdst,&packet[30], 4);
         memcpy(trama.ToS, &packet[15], 1);
         memcpy(trama.proto, &packet[23],1);
         ihl = (*IPver) << 4; //ihl indica el tamano de la cabecera ip min = 5 max = 15
         ihl = ihl >> 4;
         memcpy(trama.PortSrc, &packet[14+((ihl*32)/8)],2);//cabecera ethenet + ip
         memcpy(trama.PortDst, &packet[16+((ihl*32)/8)],2);//cabecera ethernet + ip +
hasta portdst

      if(*trama.proto == 6){
         memcpy(flags, &packet[27+((ihl*32)/8)],1);
         //extraer bit syn de flag
         syn_aux = (int)*flags & 2; //selecciono con un AND el bit de SYN (A)
         trama.syn = (syn_aux >> 1)&0x01; //lo desplazo uno a la derecha
      }else{
         trama.syn = 0;
      }

       pthread_mutex_lock(&lockTabla);
       while(addTrama(trama)==1){//mientras addTrama devuelve 1 se aumenta la tabla y
se vuelve a añadir
          tam_tabla = tam_tabla*2;
          tabla = (struct Trama *) realloc(tabla, tam_tabla);
          addTrama(trama);
       }
       pthread_mutex_unlock(&lockTabla);
      }

   }
    gettimeofday(&tv_fin, NULL);
    if(tv_fin.tv_sec-tv_ini.tv_sec>=5)
    {
       uint64_t t=((tv_fin.tv_sec*1000000)+tv_fin.tv_usec)-
((tv_ini.tv_sec*1000000)+tv_ini.tv_usec);
       fprintf(stdout,"%ld;%06ld;%ld;%06ld;%f;%f\n", tv_ini.tv_sec,
tv_ini.tv_usec,tv_fin.tv_sec, tv_fin.tv_usec,(float)count/(float)t,(float)len2/(float)t);
       len2=0;
       count=0;
       memcpy(&tv_ini,&tv_fin,sizeof(struct timeval));
       fflush(stdout);
    }

}
   pcap_close(handle);

       printf("\n\nYa se ha llenado la tabla\n\n");
```

```
//pthread_join(id,NULL);//sino no da tiepo fichero que lee desamasiado corto y hilo no
tiene tiempo a crear regla
pthread_mutex_lock(&lockEnd);
end=0;
pthread_mutex_unlock(&lockEnd);

  //se liberan recursos
  free(tabla);
  pthread_mutex_destroy(&lockEnd);
  pthread_mutex_destroy(&lockTabla);
  pthread_exit(NULL);

   return;

}


//funcion que añade una trama a la tabla...
//devuelve 0 si se guarda bien, 1 si hay que ampliar la tabla

int addTrama(struct Trama trama)
{
   int y; //*** ver que con int tienes el rango de numeros necesario para el tamaño de la
tabla sino long
   int perturb;
   int guardado = 0;
   int yini;

   //calculo de la posicion a guardar en memoria la quitupla
   y = (*trama.IPsrc + *trama.IPdst + *trama.PortSrc + *trama.PortDst +
*trama.proto)%tam_tabla;
   yini = y;

   //antes de guardar nada comprobar si la posicion esta vacia
   //Si es vacia se guarda
   while (guardado == 0)
   {
      if(vacio(tabla[y]))
      {
         guardado = 1;
         tabla[y] = trama;
         tabla[y].bitseg = trama.len;
         tabla[y].synseg = trama.syn;
      }
      else{
         //no es vacia se compara el contenido si es la misma quintupla pertenece al mismo
flujo se aumentan el bitseg
         if(compararContenido(tabla[y], trama)){
            guardado =1;
```

```
            //se actualiza synseg
            if(trama.syn == 1){
               tabla[y].synseg++;
            }

            //se actualiza bitseg
            tabla[y].bitseg = tabla[y].bitseg + trama.len;

         }else{// se recalcual el hash

            perturb = y;
            y = (5*y)+1+perturb;
            perturb >> 5;
            y = y % tam_tabla;

            //si el nuevo hash llega a ser igual que el inicial (yini) se amplia la tabla
            if(y == yini){
               //ampliar tabla
               return 1;
            }

         }
      }
   }

   return 0;

   }


int vacio(struct Trama  trama)
{
   int vacio = 1;
   uint8_t ceros[16]={0};


   if( memcmp( trama.IPdst, ceros, 16 ) ||
         memcmp( trama.IPsrc, ceros, 16 ) ||
         memcmp( trama.PortDst, ceros, 2) ||
         memcmp( trama.PortSrc, ceros, 2) ||
         memcmp( trama.proto, ceros, 1) ||
         trama.syn != 0){

      vacio = 0;

   }

   return vacio;

}
```

```
int compararContenido(struct Trama tramaTabla, struct Trama trama)
{
   int igual = 1;
   //si coinciden estos elementos es la misma quintupla y aumentamos el numero de
elementos de esa quintupla en 1 ¿que no se si es el numero de syns?
   if( memcmp( trama.IPdst, tramaTabla.IPdst, 4 ) ||
         memcmp( trama.IPsrc, tramaTabla.IPsrc, 4 ) ||
         memcmp( trama.PortDst, tramaTabla.PortDst, 2) ||
         memcmp( trama.PortSrc, tramaTabla.PortSrc, 2) ||
         memcmp( trama.proto, tramaTabla.proto, 1)){

      igual = 0;

   }

   return igual;
}
```

Anexo III: Código escenario de prueba Mininet

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

    info( '*** Adding controller\n' )
    c1=net.addController(name='c1',
                         controller=RemoteController,
                         ip='127.0.0.1',
                         port=6634)

    c0=net.addController(name='c0',
                         controller=RemoteController,
                         ip='127.0.0.1',
                         port=6633)

    info( '*** Add switches\n')
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    h1 = net.addHost('h1', cls=Host, ip='192.168.1.26', defaultRoute=None)
    h7 = net.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
    h6 = net.addHost('h6', cls=Host, ip='192.168.1.3', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host, ip='192.168.1.24', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host, ip='192.168.1.21', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host, ip='192.168.1.1', defaultRoute=None)
    h5 = net.addHost('h5', cls=Host, ip='192.168.1.2', defaultRoute=None)

    info( '*** Add links\n')
    h7s2 = {'bw':100}
    net.addLink(h7, s2, link=TCLink , **h7s2)
```

```
    s2s1 = {'bw':100}
    net.addLink(s2, s1, link=TCLink , **s2s1)
    s3s2 = {'bw':100}
    net.addLink(s3, s2, link=TCLink , **s3s2)
    h1s1 = {'bw':100}
    net.addLink(h1, s1, link=TCLink , **h1s1)
    h3s1 = {'bw':100}
    net.addLink(h3, s1, link=TCLink , **h3s1)
    h5s3 = {'bw':100}
    net.addLink(h5, s3, link=TCLink , **h5s3)
    h6s3 = {'bw':100}
    net.addLink(h6, s3, link=TCLink , **h6s3)
    h4s2 = {'bw':100}
    net.addLink(h4, s2, link=TCLink , **h4s2)
    h2s1 = {'bw':100}
    net.addLink(h2, s1, link=TCLink , **h2s1)

    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s2').start([c0])
    net.get('s3').start([c1])
    net.get('s1').start([c1])

    info( '*** Configuring switches\n')

    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```